11

Regionen in Binärbildern

Binärbilder, mit denen wir uns bereits im vorhergehenden Kapitel ausführlich beschäftigt haben, sind Bilder, in denen ein Pixel einen von nur zwei Werten annehmen kann. Wir bezeichnen diese beiden Werte häufig als „Vordergrund“ bzw. „Hintergrund“, obwohl eine solche eindeutige Unterscheidung in natürlichen Bildern oft nicht möglich ist. In diesem Kapitel gilt unser Augenmerk zusammenhängenden Bildstrukturen und insbesondere der Frage, wie wir diese isolieren und beschreiben können.

Angenommen, wir müssten ein Programm erstellen, das die Anzahl und Art der in Abb. 11.1 abgebildeten Objekte interpretiert. Solange wir jedes einzelne Pixel isoliert betrachten, werden wir nicht herausfinden, wie viele Objekte überhaupt in diesem Bild sind, wo sie sich befinden und welche Pixel zu welchem der Objekte gehören. Unsere erste Aufgabe ist daher, zunächst einmal jedes einzelne Objekt zu finden, indem wir alle Pixel zusammenfügen, die Teil dieses Objekts sind. Im einfachsten

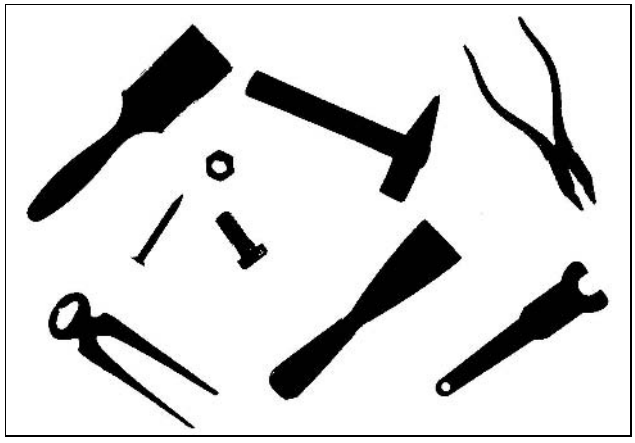


Abbildung 11.1
Binärbild mit 9 Objekten. Jedem Objekt entspricht jeweils eine zusammenhängende Region von Vordergrundpixel.

Fall ist ein Objekt eine Gruppe von aneinander angrenzenden Vordergrundpixeln bzw. eine *verbundene binäre Bildregion*.

11.1 Auffinden von Bildregionen

Bei der Suche nach binären Bildregionen sind die zunächst wichtigsten Aufgaben, herauszufinden, welche Pixel zu welcher Region gehören, wie viele Regionen im Bild existieren und wo sich diese Regionen befinden. Diese Schritte werden üblicherweise in *einem* Prozess durchgeführt, der als „Regionenmarkierung“ (*region labeling* oder auch *region coloring*) bezeichnet wird. Dabei werden zueinander benachbarte Pixel schrittweise zu Regionen zusammengefügt und allen Pixeln innerhalb einer Region eindeutige Identifikationsnummern („labels“) zugeordnet. Im Folgenden beschreiben wir zwei Varianten dieser Idee: Die erste Variante (Regionenmarkierung durch *flood filling*) füllt, ausgehend von einem gegebenen Startpunkt, jeweils eine einzige Region in alle Richtungen. Bei der zweiten Methode (*sequentielle Regionenmarkierung*) wird im Gegensatz dazu das Bild von oben nach unten durchlaufen und alle Regionen auf einmal markiert. In Abschn. 11.2.2 beschreiben wir noch ein drittes Verfahren, das die Regionenmarkierung mit dem Auffinden von Konturen kombiniert.

Unabhängig vom gewählten Ansatz müssen wir auch – durch Wahl der 4er- oder 8er-Nachbarschaft (Abb. 10.5) – fixieren, unter welchen Bedingungen zwei Pixel miteinander „verbunden“ sind, denn die beiden Arten der Nachbarschaft führen i. Allg. zu unterschiedlichen Ergebnissen. Für die Regionenmarkierung nehmen wir an, dass das zunächst binäre Ausgangsbild $I(u,v)$ die Werte 0 (Hintergrund) und 1 (Vordergrund) enthält und alle weiteren Werte für Markierungen, d. h. zur Nummerierung der Regionen, genutzt werden können:

$$I(u,v) = \begin{cases} 0 & \text{Hintergrundpixel (\textit{background})} \\ 1 & \text{Vordergrundpixel (\textit{foreground})} \\ 2, 3, \ldots & \text{Regionenmarkierung (\textit{label})} \end{cases}$$

11.1.1 Regionenmarkierung durch *Flood Filling*

Der grundlegende Algorithmus für die Regionenmarkierung durch *Flood Filling* ist einfach: Zuerst wird im Bild ein noch unmarkiertes Vordergrundpixel gesucht, von dem aus der Rest der zugehörigen Region „gefüllt“ wird. Dazu werden, ausgehend von diesem Startpixel, alle zusammenhängenden Pixel der Region besucht und markiert, ähnlich einer Flutwelle (*flood*), die sich über die Region ausbreitet. Für die Realisierung der Fülloperation gibt es verschiedene Methoden, die sich vor allem dadurch unterscheiden, wie die noch zu besuchenden Pixelkoordinaten verwaltet werden. Wir beschreiben nachfolgend drei Realisierungen der FloodFill()-Prozedur: eine rekursive Version, eine iterative *Depth-first*- und eine iterative *Breadth-first*- Version:

```
1:  RegionLabeling(I)
        I: binary image (0 = background, 1 = foreground)
2:      Initialize m ← 2 (the value of the next label to be assigned).
3:      Iterate over all image coordinates ⟨u, v⟩.
4:          if I(u, v) = 1 then
5:              FloodFill(I, u, v, m)        ▷ use any of the 3 versions below
6:              m ← m + 1.
7:      return.
```

```
8:  FloodFill(I, u, v, label)                          ▷ Recursive Version
9:      if coordinate ⟨u, v⟩ is within image boundaries and I(u, v) = 1 then
10:         Set I(u, v) ← label
11:         FloodFill(I, u+1, v, label)
12:         FloodFill(I, u, v+1, label)
13:         FloodFill(I, u, v−1, label)
14:         FloodFill(I, u−1, v, label)
15:     return.

16: FloodFill(I, u, v, label)                          ▷ Depth-First Version
17:     Create an empty stack S
18:     Put the seed coordinate ⟨u, v⟩ onto the stack: Push(S, ⟨u, v⟩)
19:     while S is not empty do
20:         Get the next coordinate from the top of the stack:
                ⟨x, y⟩ ← Pop(S)
21:         if coordinate (x, y) is within image boundaries and I(x, y) = 1
                then
22:             Set I(x, y) ← label
23:             Push(S, ⟨x+1, y⟩)
24:             Push(S, ⟨x, y+1⟩)
25:             Push(S, ⟨x, y−1⟩)
26:             Push(S, ⟨x−1, y⟩)
27:     return.

28: FloodFill(I, u, v, label)                          ▷ Breadth-First Version
29:     Create an empty queue Q
30:     Insert the seed coordinate ⟨u, v⟩ into the queue: Enqueue(Q, (u, v))
31:     while Q is not empty do
32:         Get the next coordinate from the front of the queue:
                ⟨x, y⟩ ← Dequeue(Q)
33:         if coordinate ⟨x, y⟩ is within image boundaries and I(x, y) = 1
                then
34:             Set I(x, y) ← label
35:             Enqueue(Q, ⟨x+1, y⟩)
36:             Enqueue(Q, ⟨x, y+1⟩)
37:             Enqueue(Q, ⟨x, y−1⟩)
38:             Enqueue(Q, ⟨x−1, y⟩)
39:     return.
```


Algorithmus 11.1
Regionenmarkierung durch *Flood Filling*. Das binäre Eingangsbild I enthält die Werte 0 für Hintergrundpixel und 1 für Vordergrundpixel. Es werden noch unmarkierte Vordergrundpixel gesucht, von denen aus die zugehörige Region gefüllt wird. Die FloodFill()-Prozedur ist in drei verschiedenen Varianten ausgeführt: *rekursiv*, *depth-first* und *breadth-first*.

A. **Rekursive Regionenmarkierung:** Die rekursive Version (Alg. 11.1, Zeile 8) benutzt zur Verwaltung der noch zu besuchenden Bildkoordinaten keine expliziten Datenstrukturen, sondern verwendet dazu die lokalen Variablen der rekursiven Prozedur.[1] Durch die Nachbarschaftsbeziehung zwischen den Bildelementen ergibt sich eine Baumstruktur, deren Wurzel der Startpunkt innerhalb der Region bildet. Die Rekursion entspricht einem Tiefendurchlauf (*depth-first traversal*) [19, 32] dieses Baums und führt zu einem sehr einfachen Programmcode, allerdings ist die Rekursionstiefe proportional zur Größe der Region und daher der Stack-Speicher rasch erschöpft. Die Methode ist deshalb nur für sehr kleine Bilder anwendbar.

B. **Iteratives *Flood Filling* (*depth-first*):** Jede rekursive Prozedur kann mithilfe eines eigenen *Stacks* auch iterativ implementiert werden (Alg. 11.1, Zeile 16). Der Stack dient dabei zur Verwaltung der noch „offenen“ (d. h. noch nicht bearbeiteten) Elemente. Wie in der rekursiven Version (A) wird der Baum von Bildelementen im *Depth-first*-Modus durchlaufen. Durch den eigenen, dedizierten Stack (der dynamisch im so genannten *Heap-Memory* angelegt wird) ist die Tiefe des Baums nicht mehr durch die Größe des Aufruf-Stacks beschränkt.

C. **Iteratives *Flood Filling* (*breadth-first*):** Ausgehend vom Startpunkt werden in dieser Version die jeweils angrenzenden Pixel schichtweise, ähnlich einer Wellenfront expandiert (Alg. 11.1, Zeile 28). Als Datenstruktur für die Verwaltung der noch unbearbeiteten Pixelkoordinaten wird (anstelle des Stacks) eine *Queue* (Warteschlange) verwendet. Ansonsten ist das Verfahren identisch zu Version B.

Java-Implementierung

Die rekursive Version (A) des Algorithmus ist in Java praktisch 1:1 umzusetzen. Allerdings erlaubt ein normales Java-Laufzeitsystem nicht mehr als ungefähr 10.000 rekursive Aufrufe der FloodFill()-Prozedur (Alg. 11.1, Zeile 8), bevor der Speicherplatz des Aufruf-Stacks erschöpft ist. Das reicht nur für relativ kleine Bilder mit weniger als ca. 200×200 Pixel.

Prog. 11.1 zeigt die vollständige Java-Implementierung beider Varianten der iterativen FloodFill()-Prozedur. Zunächst wird in Zeile 1 eine neue Java-Klasse `Node` zur Repräsentation einzelner Pixelkoordinaten definiert.

Zur Implementierung des Stacks in der iterativen *Depth-first*-Version (B) verwenden wir als Datenstruktur die Java-Klasse `Stack` (Prog. 11.1, Zeile 8), ein Container für beliebige Java-Objekte. Für die *Queue*-

[1] Für lokale Variablen wird in Java (oder auch in C und C++) bei jedem Prozeduraufruf der entsprechende Speicherplatz dynamisch im so genannten *Stack Memory* angelegt.

```
1 class Node {
2   int x, y;
3   Node(int x, int y) { //constructor method
4     this.x = x;  this.y = y;
5   }
6 }
```

Depth-first-Variante (mit *Stack*):

```
7 void floodFill(ImageProcessor ip, int x, int y, int label) {
8   Stack<Node> s = new Stack<Node>(); // Stack
9   s.push(new Node(x,y));
10  while (!s.isEmpty()){
11    Node n = s.pop();
12    if ((n.x>=0) && (n.x<width) && (n.y>=0) && (n.y<height)
13      && ip.getPixel(n.x,n.y)==1) {
14      ip.putPixel(n.x,n.y,label);
15      s.push(new Node(n.x+1,n.y));
16      s.push(new Node(n.x,n.y+1));
17      s.push(new Node(n.x,n.y-1));
18      s.push(new Node(n.x-1,n.y));
19    }
20  }
21 }
```

Breadth-first-Variante (mit *Queue*):

```
22 void floodFill(ImageProcessor ip, int x, int y, int label) {
23   LinkedList<Node> q = new LinkedList<Node>(); // Queue
24   q.addFirst(new Node(x,y));
25   while (!q.isEmpty()) {
26     Node n = q.removeLast();
27     if ((n.x>=0) && (n.x<width) && (n.y>=0) && (n.y<height)
28       && ip.getPixel(n.x,n.y)==1) {
29       ip.putPixel(n.x,n.y,label);
30       q.addFirst(new Node(n.x+1,n.y));
31       q.addFirst(new Node(n.x,n.y+1));
32       q.addFirst(new Node(n.x,n.y-1));
33       q.addFirst(new Node(n.x-1,n.y));
34     }
35   }
36 }
```

Programm 11.1
Flood Filling (Java-Implementierung). Die *Depth-first*-Variante verwendet als Datenstruktur die Java-Klasse `Stack` mit den Methoden `push()`, `pop()` und `isEmpty()`.

Datenstruktur in der *Breadth-first*-Variante (C) verwenden wir die Java-Klasse `LinkedList`[2] mit den Methoden `addFirst()`, `removeLast()` und `isEmpty()` (Prog. 11.1, Zeile 23). Beide Container-Klassen sind durch die Angabe `<Node>` auf den Objekttyp `Node` parametrisiert, d. h. sie können nur Objekte dieses Typs aufnehmen.[3]

Abb. 11.2 illustriert den Ablauf der Regionenmarkierung in beiden Varianten anhand eines konkreten Beispiels mit einer Region, wobei der Startpunkt („seed point“) – der normalerweise am Rand der Kontur liegt – willkürlich im Inneren der Region gewählt wurde. Deutlich ist zu sehen, dass die *Depth-first*-Methode zunächst entlang *einer* Richtung (in diesem Fall horizontal nach links) bis zum Ende der Region vorgeht und erst dann die übrigen Richtungen berücksichtigt. Im Gegensatz dazu breitet sich die Markierung bei der *Breadth-first*-Methode annähernd gleichförmig, d. h. Schicht um Schicht, in alle Richtungen aus.

Generell ist der Speicherbedarf bei der *Breadth-first*-Variante des *Flood-fill*-Verfahrens deutlich niedriger als bei der *Depth-first*-Variante. Für das Beispiel in Abb. 11.2 mit der in Prog. 11.1 gezeigten Implementierung erreicht die Größe des *Stacks* in der *Depth-first*-Variante 28.822 Elemente, während die *Queue* der *Breadth-first*-Variante nur maximal 438 Knoten aufnehmen muss.

11.1.2 Sequentielle Regionenmarkierung

Die sequentielle Regionenmarkierung ist eine klassische, nicht rekursive Technik, die in der Literatur auch als „region labeling“ bekannt ist. Der Algorithmus besteht im Wesentlichen aus zwei Schritten: (1) einer vorläufigen Markierung der Bildregionen und (2) der Auflösung von mehrfachen Markierungen innerhalb derselben Region. Das Verfahren ist (vor allem im 2. Schritt) relativ komplex, aber wegen seines moderaten Speicherbedarfs durchaus verbreitet, bietet in der Praxis allerdings gegenüber einfacheren Methoden kaum Vorteile. Der gesamte Ablauf ist in Alg. 11.2 dargestellt.

Schritt 1: Vorläufige Markierung

Im ersten Schritt des „region labeling“ wird das Bild sequentiell von links oben nach rechts unten durchlaufen und dabei jedes Vordergrundpixel mit einer vorläufigen Markierung versehen. Je nach Definition der Nachbarschaftsbeziehung werden dabei für jedes Pixel (u, v) die Umgebungen

$$\mathcal{N}_4(u,v) = \begin{array}{|c|c|c|}\hline & N_2 & \\ \hline N_1 & \times & \\ \hline & & \\ \hline\end{array} \qquad \text{oder} \qquad \mathcal{N}_8(u,v) = \begin{array}{|c|c|c|}\hline N_2 & N_3 & N_4 \\ \hline N_1 & \times & \\ \hline & & \\ \hline\end{array}$$

[2] Die Klasse `LinkedList` ist Teil des *Java Collection Frameworks* (s. auch Anhang 2.2).

[3] Generische Typen und die damit verbundene Möglichkeit zur Parametrisierung von Klassen gibt es erst seit Java 5 (1.5).

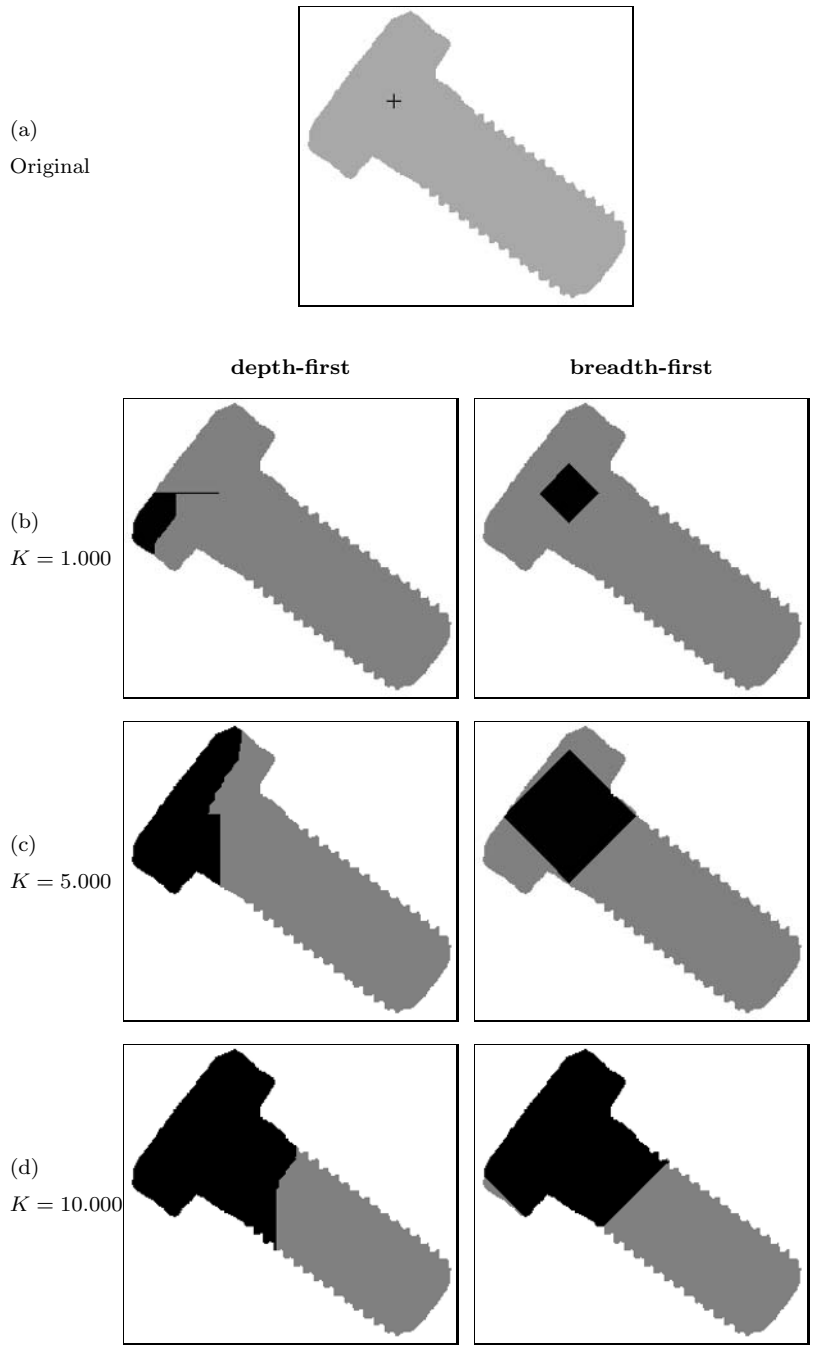


Abbildung 11.2
Iteratives *Flood Filling* – Vergleich zwischen *Depth-first*- und *Breadth-first*-Variante. Der mit + markierte Startpunkt im Originalbild (a) ist willkürlich gewählt. Zwischenergebnisse des *Flood-fill*-Algorithmus nach 1.000, 5.000 und 10.000 markierten Bildelementen (b–d). Das Bild hat eine Größe von 250×242 Pixel.

Algorithmus 11.2
Sequentielle Regionenmarkierung. Das binäre Eingangsbild I enthält die Werte $I(u,v) = 0$ für Hintergrundpixel und $I(u,v) = 1$ für Vordergrundpixel (Regionen). Die resultierenden Markierungen haben die Werte $2 \ldots m-1$.

1: SequentialLabeling(I)
I: binary image (0 = *background*, 1 = *foreground*)
2: Pass 1 – Assign Initial Labels:
3: Initialize $m \leftarrow 2$ (the value of the next label to be assigned).
4: Create an empty set $\mathcal{C}$ to hold the collisions: $\mathcal{C} \leftarrow \{\}$.
5: **for** $v \leftarrow 0 \ldots H\text{–}1$ **do** ▷ H = height of image I
6: **for** $u \leftarrow 0 \ldots W\text{–}1$ **do** ▷ W = width of image I
7: **if** $I(u,v) = 1$ **then** do one of:
8: **if** all neighbors are background pixels (all $n_i = 0$) **then**
9: $I(u,v) \leftarrow m$.
10: $m \leftarrow m + 1$.
11: **else if** exactly *one* of the neighbors has a label value $n_k > 1$ **then**
12: set $I(u,v) \leftarrow n_k$
13: **else if** *several* neighbors have label values $n_j > 1$ **then**
14: Select one of them as the new label:
$I(u,v) \leftarrow k \in \{n_j\}$.
15: **for** all other neighbors with label values $n_i > 1$ and $n_i \neq k$ **do**
16: register the pair $\langle n_i, k\rangle$ as a label collision:
$\mathcal{C} \leftarrow \mathcal{C} \cup \{\langle n_i, k\rangle\}$.

Remark: The image I now contains label values $0, 2, \ldots m\text{–}1$.

17: Pass 2 – Resolve Label Collisions:
18: Let $\mathcal{L} = \{2, 3, \ldots m\text{–}1\}$ be the set of preliminary region labels.
19: Create a partitioning of $\mathcal{L}$ as a *vector of sets*, one set for each label value: $\mathcal{R} \leftarrow [\mathcal{R}_2, \mathcal{R}_3, \ldots, \mathcal{R}_{m-1}] = [\{2\}, \{3\}, \{4\}, \ldots, \{m\text{–}1\}]$, so $\mathcal{R}_i = \{i\}$ for all $i \in \mathcal{L}$.
20: **for all** collisions $\langle a, b\rangle \in \mathcal{C}$ **do**
21: Find in $\mathcal{R}$ the sets $\mathcal{R}_a$, $\mathcal{R}_b$ containing the labels a, b, resp.:
$\mathcal{R}_a \leftarrow$ the set which currently contains label a
$\mathcal{R}_b \leftarrow$ the set which currently contains label b
22: **if** $\mathcal{R}_a \neq \mathcal{R}_b$ (a and b are contained in different sets) **then**
23: Merge sets $\mathcal{R}_a$ and $\mathcal{R}_b$ by moving all elements of $\mathcal{R}_b$ to $\mathcal{R}_a$:
$\mathcal{R}_a \leftarrow \mathcal{R}_a \cup \mathcal{R}_b$
$\mathcal{R}_b \leftarrow \{\}$

Remark: All equivalent label values (i.e., all labels of pixels in the same region) are now contained in the same sets within $\mathcal{R}$.

24: Pass 3: Relabel the Image:
25: Iterate through all image pixels (u,v):
26: **if** $I(u,v) > 1$ **then**
27: Find the set $\mathcal{R}_i$ in $\mathcal{R}$ which contains label $I(u,v)$.
28: Choose one unique, representative element k from the set $\mathcal{R}_i$ (e.g., the minimum value, $k \leftarrow \min(\mathcal{S})$).
29: Replace the image label: $I(u,v) \leftarrow k$.
30: **return** the labeled image I.

für eine 4er- bzw. 8er-Nachbarschaft berücksichtigt ($\times$ markiert das aktuelle Pixel an der Position (u, v)). Im Fall einer 4er-Nachbarschaft werden also nur die beiden Nachbarn $N_1 = I(u-1, v)$ und $N_2 = I(u, v-1)$ untersucht, bei einer 8er-Nachbarschaft die insgesamt vier Nachbarn $N_1 \ldots N_4$. Wir verwenden für das nachfolgende Beispiel eine 8er-Nachbarschaft und das Ausgangsbild in Abb. 11.3 (a).

Fortpflanzung von Markierungen

Wir nehmen an, dass die Bildwerte $I(u, v) = 0$ Hintergrundpixel und die Werte $I(u, v) = 1$ Vordergrundpixel darstellen. Nachbarpixel, die außerhalb der Bildmatrix liegen, werden als Hintergrund betrachtet. Die Nachbarschaftsregion $\mathcal{N}(u, v)$ wird nun in horizontaler und anschließend vertikaler Richtung über das Bild geschoben, ausgehend von der linken, oberen Bildecke. Sobald das aktuelle Bildelement $I(u, v)$ ein Vordergrundpixel ist, erhält es entweder eine neue Regionsnummer, oder es wird – falls bereits einer der zuvor besuchten Nachbarn in $\mathcal{N}(u, v)$ auch ein Vordergrundpixel ist – dessen Regionsnummer übernommen. Bestehende Regionsnummern (Markierungen) breiten sich dadurch von links nach rechts bzw. von oben nach unten im Bild aus (Abb. 11.3 (b–c)).

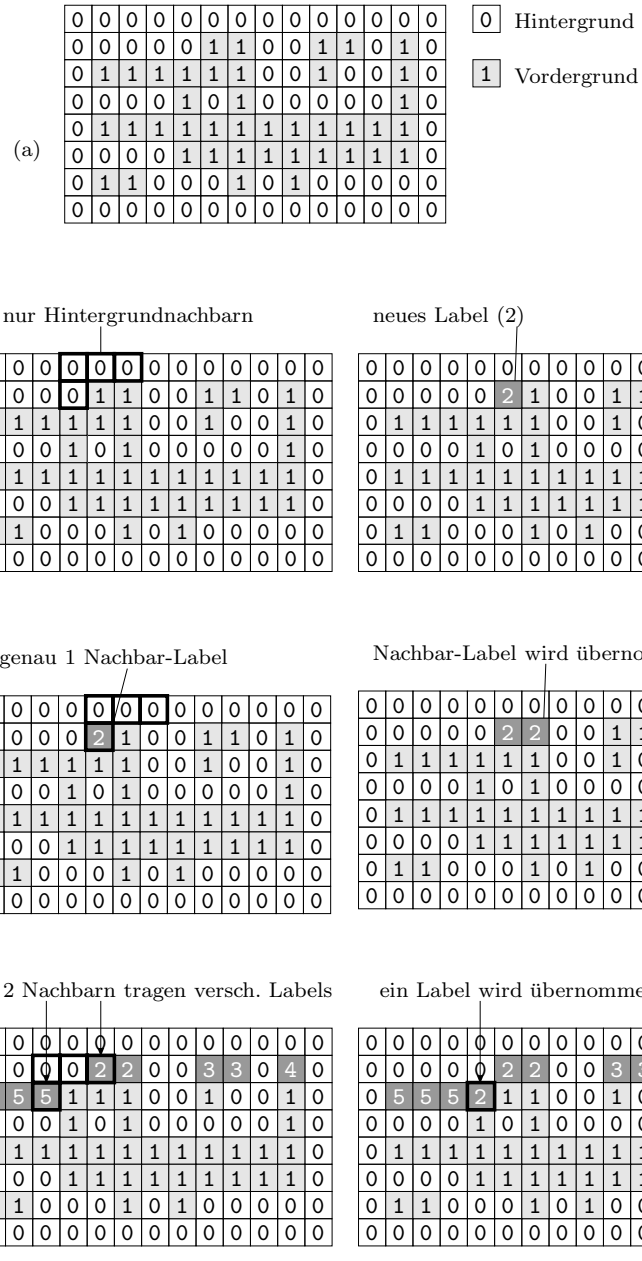
Kollidierende Markierungen

Falls zwei (oder mehr) Nachbarn bereits zu *verschiedenen* Regionen gehören, besteht eine Kollision von Markierungen, d. h., Pixel innerhalb einer zusammenhängenden Region tragen unterschiedliche Markierungen. Zum Beispiel erhalten bei einer U-förmigen Region die Pixel im linken und rechten Arm anfangs unterschiedliche Markierungen, da zunächst ja nicht sichtbar ist, dass sie tatsächlich zu einer gemeinsamen Region gehören. Die beiden Markierungen werden sich in der Folge unabhängig nach unten fortpflanzen und schließlich im unteren Teil des U kollidieren (Abb. 11.3 (d)).

Wenn zwei Markierungen a, b zusammenstoßen, wissen wir, dass sie „äquivalent“ sind und die beiden zugehörigen Bildregionen verbunden sind, also tatsächlich eine gemeinsame Region bilden. Diese Kollissionen werden im ersten Schritt nicht unmittelbar behoben, sondern nur in geeigneter Form bei ihrem Auftreten „registriert“ und erst in Schritt 2 des Algorithmus behandelt. Abhängig vom Bildinhalt können nur wenige oder auch sehr viele Kollisionen auftreten und ihre Gesamtanzahl steht erst am Ende des ersten Durchlaufs fest. Zur Verwaltung der Kollisionen benötigt man daher dynamischer Datenstrukturen, wie z. B. verkettete Listen oder *Hash*-Tabellen.

Als Ergebnis des ersten Schritts sind alle ursprünglichen Vordergrundpixel durch eine vorläufige Markierung ersetzt und die aufgetretenen Kollisionen zwischen zusammengehörigen Markierungen sind in einer geeigneten Form registriert. Das Beispiel in Abb. 11.4 zeigt das Ergebnis nach Schritt 1: alle Vordergrundpixel sind mit vorläufigen Markierungen versehen (Abb. 11.4 (a)), die aufgetretenen (als Kreise angezeigten)

Abbildung 11.3
Sequentielle Regionenmarkierung – Fortpflanzung der Markierungen. Ausgangsbild (a). Das erste Vordergrundpixel [**1**] wird in (b) gefunden: Alle Nachbarn sind Hintergrundpixel [**0**], das Pixel erhält die erste Markierung [**2**]. Im nächsten Schritt (c) ist genau *ein* Nachbarpixel mit dem Label **2** markiert, dieser Wert wird daher übernommen. In (d) sind *zwei* Nachbarpixel mit Label (**2** und **5**) versehen, einer dieser Werte wird übernommen und die Kollision $\langle \mathbf{2}, \mathbf{5} \rangle$ wird registriert.



Kollisionen zwischen Markierungen $\langle 2,4\rangle$, $\langle 2,5\rangle$ und $\langle 2,6\rangle$, wurden registriert. Die Markierungen (*labels*) $\mathcal{L} = \{2,3,4,5,6,7\}$ und Kollisionen $\mathcal{C} = \{\langle 2,4\rangle, \langle 2,5\rangle, \langle 2,6\rangle\}$ entsprechen den Knoten bzw. Kanten eines ungerichteten Graphen (Abb. 11.4 (b)).

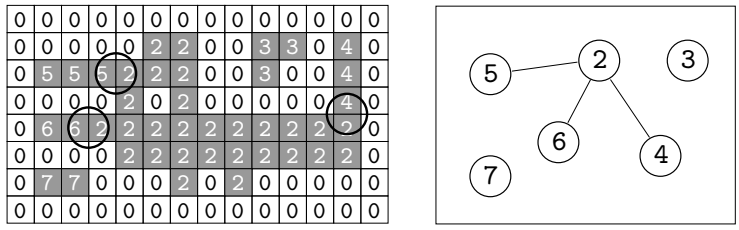


Abbildung 11.4
Sequentielle Regionenmarkierung – Ergebnis nach Schritt 1. Label-Kollisionen sind als Kreise angezeigt (a), Labels und Kollisionen entsprechen Knoten bzw. Kanten des Graphen in (b).

Schritt 2: Auflösung der Kollisionen

Aufgabe des zweiten Schritts ist die Auflösung der im ersten Schritt kollidierten Markierungen und die Verbindung der zugehörigen Teilregionen. Dieser Prozess ist nicht trivial, denn zwei unterschiedlich markierte Regionen können miteinander in transitiver Weise über eine dritte Region „verbunden“ sein bzw. im Allgemeinen über eine ganze Kette von Kollisionen. Tatsächlich ist diese Aufgabe identisch zum Problem des Auffindens zusammenhängender Teile von Graphen (*connected components problem*) [19], wobei die in Schritt 1 zugewiesenen Markierungen (*labels*) $\mathcal{L}$ den „Knoten“ des Graphen und die festgestellten Kollisionen $\mathcal{C}$ den „Kanten“ entsprechen (Abb. 11.4 (b)).

Nach dem Zusammenführen der unterschiedlichen Markierungen werden die Pixel jeder zusammenhängenden Region durch eine gemeinsame Markierung (z. B. die niedrigste der vorläufigen Markierungen innerhalb der Region) ersetzt (Abb. 11.5).

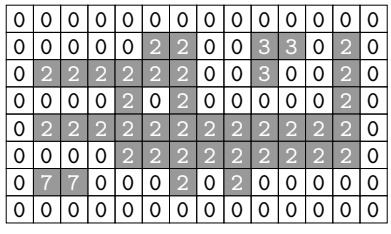


Abbildung 11.5
Sequentielle Regionenmarkierung – Endergebnis nach Schritt 2. Alle äquivalenten Markierungen wurden durch die jeweils niedrigste Markierung innerhalb einer Region ersetzt.

11.1.3 Regionenmarkierung – Zusammenfassung

Wir haben in diesem Abschnitt mehrere funktionsfähige Algorithmen zum Auffinden von zusammenhängenden Bildregionen beschrieben. Die zunächst attraktive (und elegante) Idee, die einzelnen Regionen von einem Startpunkt aus durch rekursives „flood filling“ (Abschn. 11.1.1) zu markieren, ist wegen der beschränkten Rekursionstiefe in der Praxis meist nicht anwendbar. Das klassische, sequentielle „region labeling“ (Abschn. 11.1.2) ist hingegen relativ komplex und bietet auch keinen echten Vorteil gegenüber der iterativen *Depth-first-* und *Breadth-first-*Methode, wobei letztere bei großen und komplexen Bildern generell am effektivsten ist. Abb. 11.6 zeigt ein Beispiel für eine fertige Regionenmarkierung anhand des Ausgangsbilds aus Abb. 11.1.

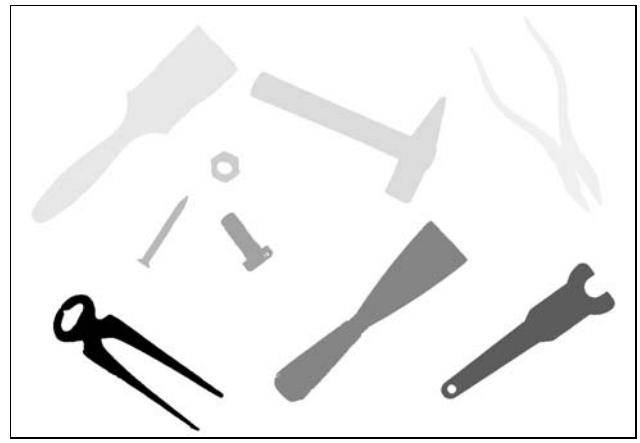


Abbildung 11.6
Beispiel für eine fertige Regionenmarkierung. Die Pixel innerhalb jeder der Region sind auf den zugehörigen, fortlaufend vergebenen Markierungswert $2, 3, \ldots 10$ gesetzt und als Grauwerte dargestellt.

11.2 Konturen von Regionen

Nachdem die Regionen eines Binärbilds gefunden sind, ist der nachfolgende Schritt häufig das Extrahieren der Umrisse oder Konturen dieser Regionen. Wie vieles andere in der Bildverarbeitung erscheint diese Aufgabe zunächst nicht schwierig – man folgt einfach den Rändern einer Region. Wie wir sehen werden, erfordert die algorithmische Umsetzung aber doch eine sorgfältige Überlegung, und tatsächlich ist das Finden von Konturen eine klassische Aufgabenstellung im Bereich der Bildanalyse.

11.2.1 Äußere und innere Konturen

Wie wir bereits in Abschn. 10.2.7 gezeigt haben, kann man die Pixel an den Rändern von binären Regionen durch morphologische Operationen und Differenzbildung auf einfache Weise identifizieren. Dieses Verfahren

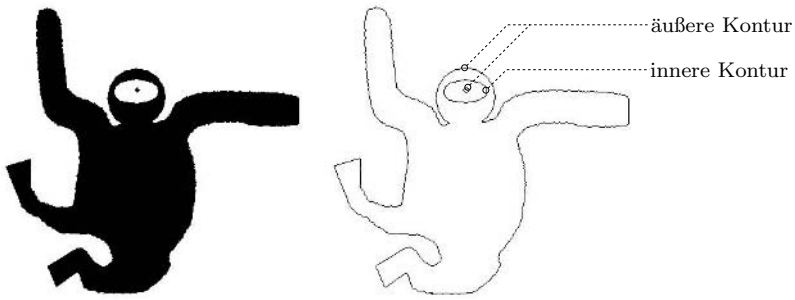


Abbildung 11.7
Binärbild mit äußeren und inneren Konturen. Äußere Konturen liegen an der Außenseite von Vordergrundregionen (dunkel). Innere Konturen umranden die Löcher von Regionen, die rekursiv weitere Regionen enthalten können.

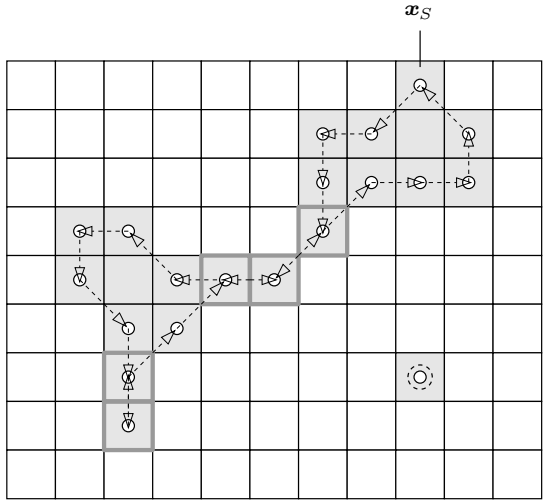


Abbildung 11.8
Pfad entlang einer Kontur als geordnete Folge von Pixelkoordinaten, ausgehend von einem beliebigen Startpunkt $\boldsymbol{x}_S$. Pixel können im Pfad mehrfach enthalten sein und auch Regionen, die nur aus einem isolierten Pixel bestehen (rechts unten), besitzen eine Kontur.

markiert die Pixel entlang der Konturen und ist z. B. für die Darstellung nützlich. Hier gehen wir jedoch einen Schritt weiter und bestimmen die Kontur jeder Region als *geordnete Folge* ihrer Randpixel. Zu beachten ist dabei, dass zusammengehörige Bildregionen zwar nur eine *äußere* Kontur aufweisen können, jedoch – innerhalb von Löchern – auch beliebig viele *innere* Konturen besitzen können. Innerhalb dieser Löcher können sich wiederum kleinere Regionen mit zugehörigen äußeren Konturen befinden, die selbst wieder Löcher aufweisen können, usw. (Abb. 11.7). Eine weitere Komplikation ergibt sich daraus, dass sich Regionen an manchen Stellen auf die Breite eines einzelnen Pixels verjüngen können, ohne ihren Zusammenhalt zu verlieren, sodass die zugehörige Kontur dieselben Pixel mehr als einmal in unterschiedlichen Richtungen durchläuft (Abb. 11.8). Wird daher eine Kontur von einem Startpunkt $\boldsymbol{x}_S$ beginnend durchlaufen, so reicht es i. Allg. nicht aus, nur wieder bis zu diesem Startpunkt zurückzukehren, sondern es muss auch die aktuelle Konturrichtung beachtet werden.

Eine Möglichkeit zur Bestimmung der Konturen besteht darin, zunächst – wie im vorherigen Abschnitt (11.1) beschrieben – die zusammengehörigen Vordergrundregionen zu identifizieren und anschließend die äußere Kontur jeder gefundenen Region zu umrunden, ausgehend von einem beliebigen Randpixel der Region. In ähnlicher Weise wären dann die inneren Konturen aus den Löchern der Regionen zu ermitteln. Für diese Aufgabe gibt es eine Reihe von Algorithmen, wie beispielsweise in [71], [64, S. 142–148] oder [76, S. 296] beschrieben.

Als Alternative zeigen wir nachfolgend ein *kombiniertes* Verfahren, das im Unterschied zum traditionellen Ansatz die Konturfindung und die Regionenmarkierung verbindet.

11.2.2 Kombinierte Regionenmarkierung und Konturfindung

Dieses Verfahren aus [17] verbindet Konzepte der sequentiellen Regionenmarkierung (Abschn. 11.1) und der traditionellen Konturverfolgung, um in einem Bilddurchlauf beide Aufgaben gleichzeitig zu erledigen. Es werden sowohl äußere wie innere Konturen sowie die zugehörigen Regionen identifiziert und markiert. Der Algorithmus benötigt keine komplizierten Datenstrukturen und ist im Vergleich zu ähnlichen Verfahren sehr effizient.

Die grundlegende Idee des Algorithmus ist einfach und nachfolgend skizziert. Demgegenüber ist die konkrete Realisierung im Detail relativ aufwendig und daher als vollständige Implementierung als Java-Programm bzw. als ImageJ-Plugin in Anhang 4.2 (S. 487–496) aufgelistet. Die wichtigsten Schritte des Verfahrens sind in Abb. 11.9 illustriert:

1. Das Binärbild I wird – ähnlich wie bei der sequentiellen Regionenmarkierung (Alg. 11.2) – von links oben nach rechts unten durchlaufen. Damit ist sichergestellt, dass alle Pixel im Bild berücksichtigt werden und am Ende eine entsprechende Markierung tragen.
2. An der aktuellen Bildposition können folgende Situationen auftreten:

   Fall A: Der Übergang von einem Hintergrundpixel auf ein bisher nicht markiertes Vordergrundpixel A bedeutet, dass A am Außenrand einer neuen Region liegt. Eine neue Marke (*label*) wird erzeugt und die zugehörige *äußere* Kontur wird (durch die Prozedur TraceContour in Alg. 11.3) umfahren und markiert (Abb. 11.9 (a)). Zudem werden auch alle unmittelbar angrenzenden Hintergrundpixel (mit dem Wert -1) markiert.

   Fall B: Der Übergang von einem Vordergrundpixel B auf ein nicht markiertes Hintergrundpixel bedeutet, dass B am Rand einer *inneren* Kontur liegt (Abb. 11.9 (b)). Ausgehend von B wird die innere Kontur umfahren und deren Pixel mit der Markierung der einschließenden Region versehen (Abb. 11.9 (c)). Auch alle angrenzenden Hintergrundpixel werden wiederum (mit dem Wert -1) markiert.

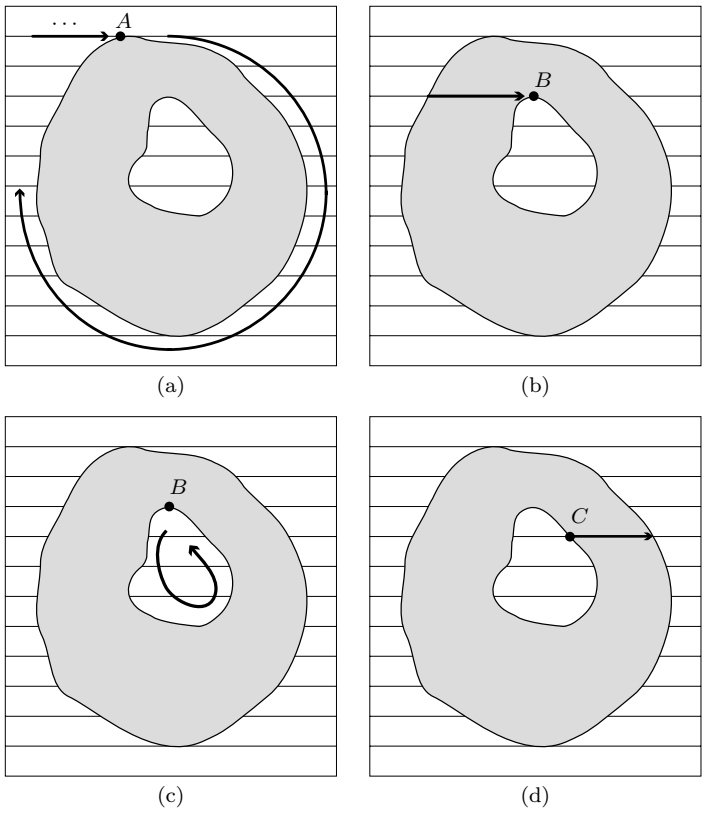


Abbildung 11.9
Kombinierte Regionenmarkierung und Konturverfolgung (nach [17]). Das Bild wird von links oben nach rechts unten zeilenweise durchlaufen. In (a) ist der erste Punkt A am äußeren Rand einer Region gefunden. Ausgehend von A werden die Randpixel entlang der äußeren Kontur besucht und markiert, bis A wieder erreicht ist. In (b) ist der erste Punkt B auf einer inneren Kontur gefunden. Die innere Kontur wird wiederum bis zum Punkt B zurück durchlaufen und markiert (c). In (d) wird ein bereits markierter Punkt C auf einer inneren Kontur gefunden. Diese Markierung wird entlang der Bildzeile innerhalb der Region fortgepflanzt.

Fall C: Bei einem Vordergrundpixel, das nicht an einer Kontur liegt, ist jedenfalls das linke Nachbarpixel bereits markiert (Abb. 11.9 (d)). Diese Markierung wird für das aktuelle Pixel übernommen.

In Alg. 11.3–11.4 ist der gesamte Vorgang nochmals exakt beschrieben. Die Prozedur CombinedContourLabeling durchläuft das Bild zeilenweise und ruft die Prozedur TraceContour auf, sobald eine neue innere oder äußere Kontur zu bestimmen ist. Die Markierungen der Bildelemente entlang der Konturen sowie der umliegenden Hintergrundpixel werden im „label map“ *LM* durch die Methode FindNextNode (Alg. 11.4) eingetragen.

11.2.3 Implementierung

Die vollständige Implementierung in Java bzw. ImageJ ist aus Platzgründen in Anhang 4.2 (S. 487–496) zusammengestellt. Sie folgt im Wesentlichen der Beschreibung in Alg. 11.3–11.4, weist jedoch einige zusätzliche Details auf:[4]

[4] Nachfolgend sind zu den im Algorithmus verwendeten Symbolen jeweils die Bezeichnungen im Java-Programm in Klammern angegeben.

Algorithmus 11.3
Kombinierte Konturfindung und Regionenmarkierung. Die Prozedur CombinedContourLabeling erzeugt aus dem Binärbild I eine Menge von Konturen sowie ein Array mit der Regionenmarkierung aller Bildpunkte. Wird ein neuer Konturpunkt (äußere oder innere Kontur) gefunden, dann wird die eigentliche Kontur als Folge von Konturpunkten durch den Aufruf von TraceContour (Zeile 20 bzw. Zeile 27) ermittelt. TraceContour selbst ist in Alg. 11.4 beschrieben.

```
1:  CombinedContourLabeling (I)
        I: binary image
2:      Create an empty set of contours: C ← {}
3:      Create a label map LM of the same size as I and initialize:
4:      for all (u, v) do
5:          LM(u, v) ← 0                                  ▷ label map LM
6:      R ← 0                                             ▷ region counter R
7:      Scan the image from left to right, top to bottom:
8:      for v ← 0 ... N−1 do
9:          L_c ← 0                                       ▷ current label L_c
10:         for u ← 0 ... M−1 do
11:             if I(u, v) is a foreground pixel then
12:                 if (L_c ≠ 0) then                     ▷ continue existing region
13:                     LM(u, v) ← L
14:                 else
15:                     L_c ← LM(u, v)
16:                     if (L_c = 0) then                 ▷ hit new outer contour
17:                         R ← R + 1
18:                         L_c ← R
19:                         x_S ← (u, v)
20:                         C_outer ← TraceContour(x_S, 0, L_c, I, LM)
21:                         C ← C ∪ {C_outer}             ▷ collect new contour
22:                         LM(u, v) ← L_c
23:             else                                      ▷ I(u, v) is a background pixel
24:                 if (L ≠ 0) then
25:                     if (LM(u, v) = 0) then            ▷ hit new inner contour
26:                         x_S ← (u−1, v)
27:                         C_inner ← TraceContour(x_S, 1, L_c, I, LM)
28:                         C ← C ∪ {C_inner}             ▷ collect new contour
29:                     L ← 0
30:     return (C, LM).                                   ▷ return the set of contours and the label map
```

Fortsetzung in Alg. 11.4 ▷▷

- Zunächst wird das Bild I (`pixelMap`) und das zugehörige *label map* LM (`labelMap`) an allen Rändern um ein zusätzliches Pixel vergrößert, wobei im Bild I diese Pixel als Hintergrund markiert werden. Dies vereinfacht die Verfolgung der Konturen, da bei der Behandlung der Ränder nun keine besonderen Vorkehrungen mehr notwendig sind.
- Die gefundenen Konturen werden in einem Objekt der Klasse `ContourSet` zusammengefasst, und zwar getrennt in äußere und innere Konturen. Diese sind wiederum Objekte der Klassen `OuterContour` bzw. `InnerContour` mit der gemeinsamen Überklasse `Contour`. Jede Kontur besteht aus einer geordneten Folge von Koordinatenpunkten der Klasse `Node` (Definition auf S. 199). Als dynamische Datenstruktur für die Koordinatenpunkte sowie für die Menge der äußeren und

Algorithmus 11.4
Kombinierte Konturfindung und Regionenmarkierung (*Fortsetzung*). Die Prozedur TraceContour durchläuft die zum Startpunkt $\boldsymbol{x}_S$ gehörigen Konturpunkte, beginnend mit der Suchrichtung $d_S = 0$ (äußere Kontur) oder $d_S = 1$ (innere Kontur). Dabei werden alle Konturpunkte sowie benachbarte Hintergrundpunkte im Label-Array LM markiert. TraceContour verwendet FindNextNode(), um zu einem gegebenen Punkt $\boldsymbol{x}_c$ den nachfolgenden Konturpunkt zu bestimmen (Zeile 10). Die Funktion Delta() dient lediglich zur Bestimmung der Folgekoordinaten in Abhängigkeit von der aktuellen Suchrichtung d.

```
 1: TraceContour(x_S, d_S, L_C, I, LM)
        x_S: start position
        d_S: initial search direction
        L_C: label for this contour
        I: image
        LM: label map
 2:     Create an empty contour C
 3:     (x_T, d) ← FindNextNode(x_S, d_s, I, LM)
 4:     Append(C, x_T)                          ▷ add x_T to contour C
 5:     x_p ← x_S                               ▷ previous position x_p = (u_p, v_p)
 6:     x_c ← x_T                               ▷ current position x_c = (u_c, v_c)
 7:     done ← (x_S = x_T)                      ▷ isolated pixel?
 8:     while (¬done) do
 9:         LM(u_c, v_c) ← L_C
10:         (x_n, d) ← FindNextNode(x_c, (d+6) mod 8, I, LM)
11:         x_p ← x_c
12:         x_c ← x_n
13:         done ← (x_p = x_S ∧ x_c = x_T)      ▷ back at starting position?
14:         if (¬done) then
15:             Append(C, x_n)                  ▷ add point x_n to contour C
16:     return C .                              ▷ return this contour

17: FindNextNode(x_c, d, I, LM)
        x_c: original position, d: search direction, I: image, LM: label map
18:     for i ← 0...6 do                        ▷ search in 7 directions
19:         x' ← x_c + Delta(d)                 ▷ x' = (u', v')
20:         if I(u', v') is a background pixel then
21:             LM(u', v') ← −1                 ▷ mark background as visited (−1)
22:             d ← (d + 1) mod 8
23:         else                                ▷ found a non-background pixel at x'
24:             return (x', d)
25:     return (x_c, d) .                       ▷ found no next node, return start position
```

26: Delta$(d) = (\Delta x, \Delta y)$, wobei

d	0	1	2	3	4	5	6	7
Δx	1	1	0	−1	−1	−1	0	1
Δy	0	1	1	1	0	−1	−1	−1

inneren Konturen wird die Java-Container-Klasse `ArrayList` (parametrisiert auf den Typ `Node`) verwendet.

- Die Methode `traceContour()` in Prog. 11.4 durchläuft eine äußere oder innere Kontur, beginnend bei einem Startpunkt $\boldsymbol{x}_S$ (`xS`, `yS`). Dafür wird zunächst die Methode `findNextNode()` aufgerufen, um den auf $\boldsymbol{x}_S$ folgenden Konturpunkt $\boldsymbol{x}_T$ (`xT`, `yT`) zu bestimmen:
  - Für den Fall, dass kein nachfolgender Punkt gefunden wird, gilt $\boldsymbol{x}_S = \boldsymbol{x}_T$ und es handelt sich um eine Region (Kontur), bestehend aus einem isolierten Pixel. In diesem Fall ist `traceContour()` fertig.
  - Andernfalls werden durch wiederholten Aufruf von `findNextNode()` die übrigen Konturpunkte schrittweise durchlaufen, wo-

Programm 11.2
Beispiel für die Verwendung der Klasse `ContourTracer`.

```
1 import java.util.ArrayList;
2 ...
3 public class ContourTracingPlugin_ implements PlugInFilter {
4   public void run(ImageProcessor ip) {
5     ContourTracer tracer = new ContourTracer(ip);
6     ContourSet cs = tracer.getContours();
7     // process outer and inner contours:
8     ArrayList<Contour> outer = cs.outerContours;
9     ArrayList<Contour> inner = cs.innerContours;
10    ...
11  }
12 }
```

bei jeweils ein aufeinander folgendes Paar von Punkten, der aktuelle (*current*) Punkt $\boldsymbol{x}_c$ (`xC`, `yC`) und der vorherige (*previous*) Punkt $\boldsymbol{x}_p$ (`xP`, `yP`), mitgeführt werden. Erst wenn *beide* Punkte mit den ursprünglichen Startpunkten der Kontur, $\boldsymbol{x}_S$ und $\boldsymbol{x}_T$, übereinstimmen, ist die Kontur vollständig durchlaufen.

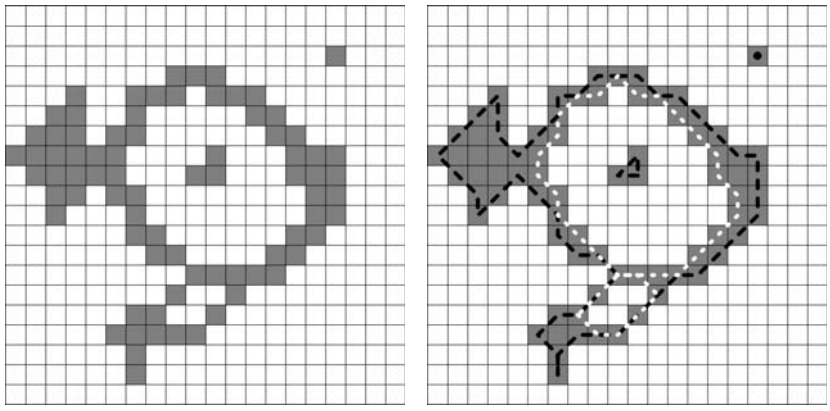
- Die Methode `findNextNode()` bestimmt den zum aktuellen Punkt $\boldsymbol{x}_c$ (`Xc`) nachfolgenden Konturpunkt, wobei die anfängliche Such*richtung* d (`dir`) von der Lage zum vorherigen Konturpunkt abhängt. Ausgehend von der ersten Suchrichtung werden maximal 7 Nachbarpunkte (alle außer dem vorherigen Konturpunkt) im Uhrzeigersinn untersucht, bis ein Vordergrundpixel (= Nachfolgepunkt) gefunden ist. Gleichzeitig werden alle gefundenen Hintergrundpixel mit dem Wert -1 im *label map LM* (`labelMap`) markiert, um wiederholte Besuche zu vermeiden. Wird unter den 7 möglichen Nachbarn kein gültiger Nachfolgepunkt gefunden, dann gibt `findNextNode()` den Ausgangspunkt $\boldsymbol{x}_c$ zurück.

Die Hauptfunktionalität ist bei dieser Implementierung in der Klasse `ContourTracer` verpackt, deren prinzipielle Verwendung innerhalb der `run()`-Methode eines ImageJ-Plugins in Prog. 11.2 gezeigt ist.

Die Implementierung in Anhang 4.2 zeigt außerdem die Darstellung der Konturen als Vektorgrafik mithilfe der Klasse `ContourOverlay`. Auf diese Weise können grafische Strukuren, die kleiner bzw. dünner sind als Bildpixel, über einem Rasterbild in ImageJ angezeigt werden.

11.2.4 Beispiele

Der kombinierte Algorithmus zur Regionenmarkierung und Konturverfolgung ist aufgrund seines bescheidenen Speichererbedarfs auch für große Binärbilder problemlos und effizient anwendbar. Abb. 11.10 zeigt ein Beispiel anhand eines vergrößerten Bildausschnitts, in dem mehrere spezielle Situationen in Erscheinung treten, wie einzelne, isolierte Pixel und Verdünnungen, die der Konturpfad in beiden Richtungen passieren



(a) (b)

Abbildung 11.10
Kombinierte Konturfindung und Regionenmarkierung. Originalbild, Vordergrundpixel sind grau markiert (a). Gefundene Konturen (b), mit schwarzen Linien für äußere und weißen Linien für innere Konturen. Die Konturen sind durch Polygone gekennzeichnet, deren Knoten jeweils im Zentrum eines Konturpixels liegen. Konturen für isolierte Pixel (z. B. rechts oben in (b)) sind durch kreisförmige Punkte dargestellt.

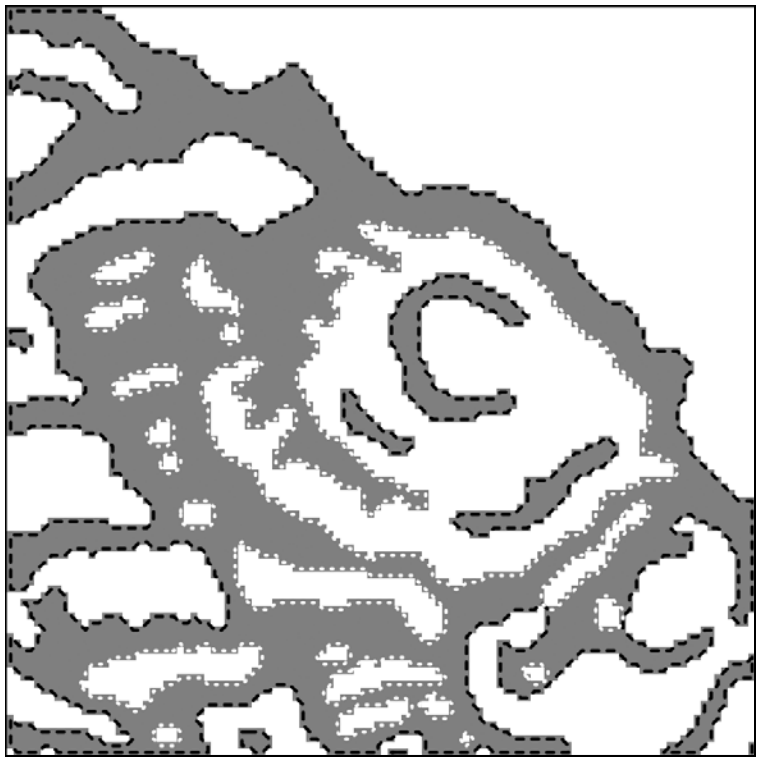


Abbildung 11.11
Bildausschnitt mit Beispielen von komplexen Konturen (Originalbild in Abb. 10.11). Äußere Konturen sind schwarz, innere Konturen weiß markiert.

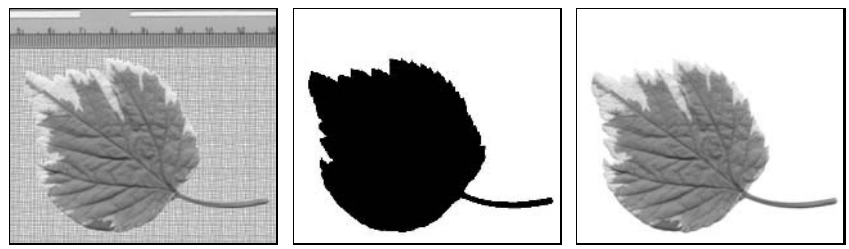
muss. Die Konturen selbst sind durch Polygonzüge zwischen den Mittelpunkten der enthaltenen Pixel dargestellt, äußere Konturen sind schwarz und innere Konturen sind weiß gezeichnet. Regionen bzw. Konturen, die nur aus einem Pixel bestehen, sind durch Kreise in der entsprechenden Farbe markiert. Abb. 11.11 zeigt das Ergebnis für einen größeren Ausschnitt aus einem konkreten Originalbild (Abb. 10.11) in der gleichen Darstellungsweise.

11.3 Repräsentation von Bildregionen

11.3.1 Matrix-Repräsentation

Eine natürliche Darstellungsform für Bilder ist eine Matrix bzw. ein zweidimensionales Array, in dem jedes Element die Intensität oder die Farbe der entsprechenden Bildposition enthält. Diese Repräsentation kann in den meisten Programmiersprachen einfach und elegant abgebildet werden und ermöglicht eine natürliche Form der Verarbeitung im Bildraster. Ein möglicher Nachteil ist, dass diese Darstellung die Struktur des Bilds nicht berücksichtigt. Es macht keinen Unterschied, ob das Bild nur ein paar Linien oder eine komplexe Szene darstellt – die erforderliche Speichermenge ist konstant und hängt nur von der Dimension des Bilds ab.

Binäre Bildregionen können mit einer logischen Maske dargestellt werden, die innerhalb der Region den Wert *true* und außerhalb den Wert *false* enthält (Abb. 11.12). Da ein logischer Wert mit nur einem Bit dargestellt werden kann, bezeichnet man eine solche Matrix häufig als „bitmap“.[5]



(a) (b) (c)

Abbildung 11.12
Verwendung einer logischen Bildmaske zu Spezifikation einer Bildregion. Originalbild (a), Bildmaske (b), maskiertes Bild (c).

11.3.2 Lauflängenkodierung

Bei der Lauflängenkodierung (*run length encoding*, RLE) werden aufeinander folgende Vordergrundpixel zu Blöcken zusammengefasst. Ein Block oder „run“ ist eine möglichst lange Folge von gleichartigen Pixeln innerhalb einer Bildzeile oder -spalte. Runs können in kompakter Form mit nur drei ganzzahligen Werten

$$Run_i = \langle row_i, column_i, length_i \rangle$$

dargestellt werden (Abb. 11.13). Werden die Runs innerhalb einer Zeile zusammengefasst, so ist natürlich die Zeilennummer redundant und kann

[5] In Java werden allerdings für Variablen vom Typ `boolean` immer 8 Bits verwendet und ein „kleinerer“ Datentyp ist nicht verfügbar. Echte „bitmaps“ können daher in Java nicht direkt realisiert werden.

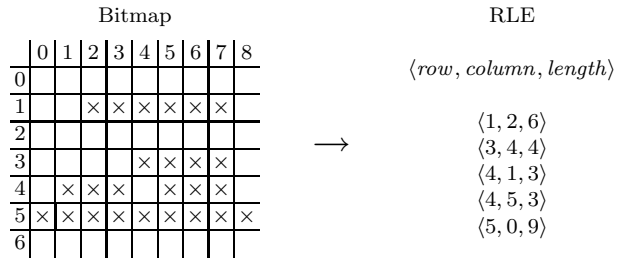


Abbildung 11.13
Lauflängenkodierung in Zeilenrichtung. Ein zusammengehöriger Pixelblock wird durch seinen Startpunkt (1, 2) und seine Länge (6) repräsentiert.

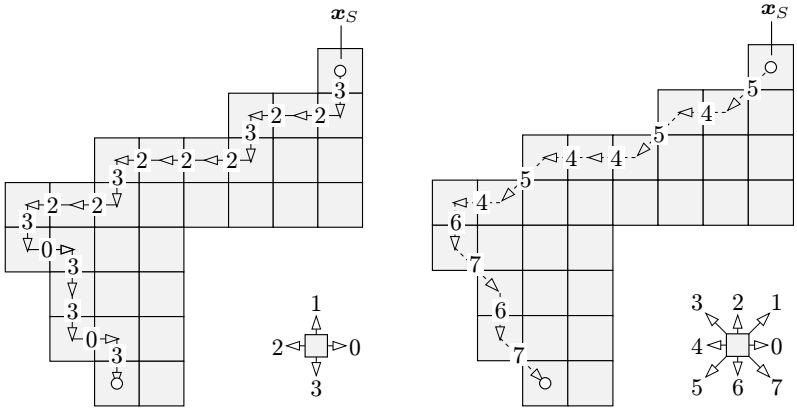
entfallen. In manchen Anwendungen kann es sinnvoll sein, die abschließende Spaltennummer anstatt der Länge des Blocks zu speichern.

Die RLE-Darstellung ist schnell und einfach zu berechnen. Sie ist auch als simple (verlustfreie) Kompressionsmethode seit langem in Gebrauch und wird auch heute noch verwendet, beispielsweise im TIFF-, GIF- und JPEG-Format sowie bei der Faxkodierung. Aus RLE-kodierten Bildern können auch statistische Eigenschaften, wie beispielsweise Momente (siehe Abschn. 11.4.3), auf direktem Weg berechnet werden.

11.3.3 *Chain Codes*

Regionen können nicht nur durch ihre innere Fläche, sondern auch durch ihre Konturen dargestellt werden. Eine klassische Form dieser Darstellung sind so genannte „Chain Codes“ oder „Freeman Codes“ [27]. Dabei wird die Kontur, ausgehend von einem Startpunkt $\boldsymbol{x}_S$, als Folge von Positionsänderungen im diskreten Bildraster repräsentiert (Abb. 11.14).

Für eine geschlossene Kontur, gegeben durch die Punktfolge $B_\mathcal{R} = (\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1})$, mit $\boldsymbol{x}_i = (u_i, v_i)$, erzeugen wir die Elemente der zugehörigen Chain-Code-Folge $C_\mathcal{R} = (c_0, c_1, \ldots c_{M-1})$ mit



4-Chain Code
3223222322303303...111
length = 28

8-Chain Code
54544546767...222
length = $18 + 5\sqrt{2} \approx 25$

Abbildung 11.14
Chain Codes mit 4er- und 8er-Nachbarschaft. Zur Berechnung des Chain Codes wird die Kontur von einem Startpunkt $\boldsymbol{x}_S$ aus durchlaufen. Die relative Position zwischen benachbarten Konturpunkten bestimmt den Richtungscode innerhalb einer 4er-Nachbarschaft (links) oder einer 8er-Nachbarschaft (rechts). Die Länge des resultierenden Pfads, berechnet aus der Summe der Einzelsegmente, ergibt eine Schätzung für die tatsächliche Konturlänge.

$$c_i = \mathit{Code}(\Delta u_i, \Delta v_i)\,, \quad \text{wobei} \tag{11.1}$$
$$(\Delta u_i, \Delta v_i) = \begin{cases} (u_{i+1}-u_i, v_{i+1}-v_i) & \text{für } 0 \le i < M-1 \\ (u_0-u_i, v_0-v_i) & \text{für } i = M-1 \end{cases}$$

und $\mathit{Code}(\Delta u_i, \Delta v_i)$ aus folgender Tabelle[6] ermittelt wird:

Δu_i	1	1	0	−1	−1	−1	0	1
Δv_i	0	1	1	1	0	−1	−1	−1
$\mathit{Code}(\Delta u_i, \Delta v_i)$	0	1	2	3	4	5	6	7

Chain Codes sind kompakt, da nur die absoluten Koordinaten für den Startpunkt und nicht für jeden Konturpunkt gespeichert werden. Zudem können die 8 möglichen relativen Richtungen zwischen benachbarten Konturpunkten mit kleineren Zahlentypen (3 Bits) kodiert werden.

Differentieller Chain Code

Ein Vergleich von zwei mit Chain Codes dargestellten Regionen ist allerdings auf direktem Weg nicht möglich. Zum einen ist die Beschreibung vom gewählten Startpunkt $\boldsymbol{x}_S$ abhängig, zum anderen führt die Drehung der Region um 90° zu einem völlig anderen Chain Code. Eine geringfügige Verbesserung bringt der *differentielle* Chain Code, bei dem nicht die Differenzen der Positionen aufeinander folgender Konturpunkte, sondern die Änderungen der Richtung entlang der diskreten Kontur kodiert werden. Aus einem *absoluten* Chain-Code $C_\mathcal{R} = (c_0, c_1, \ldots c_{M-1})$ werden die Elemente des *differentiellen* Chain Codes $C'_\mathcal{R} = (c'_0, c'_1, \ldots c'_{M-1})$ in der Form

$$c'_i = \begin{cases} (c_{i+1} - c_i) \bmod 8 & \text{für } 0 \le i < M-1 \\ (c_0 - c_i) \bmod 8 & \text{für } i = M-1 \end{cases} \tag{11.2}$$

berechnet,[7] wiederum unter Annahme der 8er-Nachbarschaft. Das Element c'_i beschreibt also die Richtungsänderung (Krümmung) der Kontur zwischen den aufeinander folgenden Segmenten c_i und c_{i+1} des ursprünglichen Chain Codes $C_\mathcal{R}$. Für die Kontur in Abb. 11.14 (b) wäre das Ergebnis

$$C_\mathcal{R} = (5, 4, 5, 4, 4, 5, 4, 6, 7, 6, 7, \ldots 2, 2, 2)$$
$$C'_\mathcal{R} = (7, 1, 7, 0, 1, 7, 2, 1, 7, 1, 1, \ldots 0, 0, 3)$$

Die ursprüngliche Kontur kann natürlich bei Kenntnis des Startpunkts $\boldsymbol{x}_S$ und der Anfangsrichtung c'_0 auch aus einem differentiellen Chain Code wieder vollständig rekonstruiert werden.

[6] Unter Verwendung der 8er-Nachbarschaft.
[7] Zur Implementierung des mod-Operators in Java siehe Anhang B.1.2.

Shape Numbers

Der differentielle Chain Code bleibt zwar bei einer Drehung der Region um 90° unverändert, ist aber weiterhin vom gewählten Startpunkt abhängig. Möchte man zwei durch ihre differentiellen Chain Codes C'_1, C'_2 gegebenen Konturen von gleicher Länge M auf ihre Ähnlichkeit untersuchen, so muss zunächst ein gemeinsamer Startpunkt festgelegt werden. Eine häufig angeführte Methode [4,30] besteht darin, die Codefolge C' als Ziffern einer Zahl (zur Basis $B = 8$ bzw. $B = 4$ bei einer 4er-Nachbarschaft) zu interpretieren, d. h. mit dem Wert

$$\mathit{Value}(C') = c'_0 \cdot B^0 + c'_1 \cdot B^1 + \ldots + c'_{M-1} \cdot B^{M-1} = \sum_{i=0}^{M-1} c_i \cdot B^i \ . \tag{11.3}$$

Dann wird die Folge C' soweit zyklisch um k Positionen verschoben, bis sich ein maximaler Wert

$$\mathit{Value}(C' \rightarrow k) = \max! \qquad \text{für } \ 0 \le k < M$$

ergibt. $C' \rightarrow k$ bezeichnet dabei die zyklisch um k Positionen nach rechts verschobene Folge C', z. B.

$$C' = (0, 1, 3, 2, \ldots 0, 3, 7, 4) \ , \qquad C' \rightarrow 2 = (7, 4, 0, 1, \ldots 0, 3) \ .$$

Die resultierende Folge („Shape Number“) ist dann bezüglich des Startpunkts „normalisiert“ und kann ohne weitere Verschiebung elementweise mit anderen normalisierten Folgen verglichen werden. Allerdings würde die Funktion $\mathit{Value}(C')$ in Gl. 11.3 viel zu große Werte erzeugen, um sie tatsächlich berechnen zu können. Einfacher ist es, die Relation

$$\mathit{Value}(C'_1) > \mathit{Value}(C'_2)$$

auf Basis der *lexikographischen Ordnung* zwischen den (Zeichen-)Folgen C'_1 und C'_2 zu ermitteln, ohne deren arithmetische Werte wirklich zu berechnen.

Der Vergleich auf Basis der Chain Codes ist jedoch generell keine besonders zuverlässige Methode zur Messung der Ähnlichkeit von Regionen, allein deshalb, weil etwa Drehungen um beliebige Winkel ($\neq 90°$) zu großen Differenzen im Code führen können. Darüber hinaus sind natürlich Größenveränderungen (Skalierungen) oder andere Verzerrungen mit diesen Methoden überhaupt nicht handhabbar. Für diese Zwecke finden sich wesentlich bessere Werkzeuge im nachfolgenden Abschnitt 11.4.

Fourierdeskriptoren

Ein eleganter Ansatz zur Beschreibung von Konturen sind so genannte „Fourierdeskriptoren“, bei denen die zweidimensionale Kontur $B_{\mathcal{R}} =$

$(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1})$ mit $\boldsymbol{x}_i = (u_i, v_i)$ als Folge von komplexen Werten $(z_0, z_1 \ldots z_{M-1})$ mit

$$z_i = (u_i + \mathrm{i} \cdot v_i) \in \mathbb{C} \tag{11.4}$$

interpretiert wird. Aus dieser Folge lässt sich (durch geeignete Interpolation) eine diskrete, eindimensionale, periodische Funktion $f(s) \in \mathbb{C}$ mit konstanten Abtastintervallen über die Konturlänge s ableiten. Die Koeffizienten des (eindimensionalen) *Fourierspektrums* (siehe Abschn. 13.3) der Funktion $f(s)$ bilden eine Formbeschreibung der Kontur im Frequenzraum, wobei die niedrigen Spektralkoeffizienten eine grobe Formbeschreibung liefern. Details zu diesem klassischen Verfahren finden sich beispielsweise in [30, 33, 48, 49, 80].

11.4 Eigenschaften binärer Bildregionen

Angenommen man müsste den Inhalt eines Digitalbilds einer anderen Person am Telefon beschreiben. Eine Möglichkeit bestünde darin, die einzelnen Pixelwerte in einer bestimmten Ordnung aufzulisten und durchzugeben. Ein weniger mühsamer Ansatz wäre, das Bild auf Basis von Eigenschaften auf einer höheren Ebene zu beschreiben, etwa als „ein rotes Rechteck auf einem blauen Hintergrund“ oder „ein Sonnenuntergang am Strand mit zwei im Sand spielenden Hunden“ usw. Während uns so ein Vorgehen durchaus natürlich und einfach erscheint, ist die Generierung derartiger Beschreibungen für Computer ohne menschliche Hilfe derzeit (noch) nicht realisierbar. Für den Computer einfacher ist die Berechnung mathematischer Eigenschaften von Bildern oder einzelner Bildteile, die zumindest eine eingeschränkte Form von Klassifikation ermöglichen. Dies wird als „Mustererkennung“ (*Pattern Recognition*) bezeichnet und bildet ein eigenes wissenschaftliches Fachgebiet, das weit über die Bildverarbeitung hinausgeht [21, 62, 83].

11.4.1 Formmerkmale (*Features*)

Der Vergleich und die Klassifikation von binären Regionen ist ein häufiger Anwendungsfall, beispielsweise bei der „optischen“ Zeichenerkennung (*optical character recognition*, OCR), beim automatischen Zählen von Zellen in Blutproben oder bei der Inspektion von Fertigungsteilen auf einem Fließband. Die Analyse von binären Regionen gehört zu den einfachsten Verfahren und erweist sich in vielen Anwendungen als effizient und zuverlässig.

Als „Feature“ einer Region bezeichnet man ein bestimmtes numerisches oder qualitatives Merkmal, das aus ihren Bildpunkten (Werten und Koordinaten) berechnet wird, im einfachsten Fall etwa die Größe (Anzahl der Pixel) einer Region. Um eine Region möglichst eindeutig zu beschreiben, werden üblicherweise verschiedene Features zu einem „Feature Vector“ kombiniert. Dieser stellt gewissermaßen die „Signatur“

einer Region dar, die zur Klassifikation bzw. zur Unterscheidung gegenüber anderen Regionen dient. Features sollen einfach zu berechnen und möglichst unbeeinflusst („robust“) von nicht relevanten Veränderungen sein, insbesondere gegenüber einer räumlichen Verschiebung, Rotation oder Skalierung.

11.4.2 Geometrische Eigenschaften

Die Region $\mathcal{R}$ eines Binärbilds kann als zweidimensionale Verteilung von Vordergrundpunkten $\boldsymbol{x}_i = (u_i, v_i)$ in der diskreten Ebene $\mathbb{Z}^2$ interpretiert werden, d. h.

$$\mathcal{R} = \{\boldsymbol{x}_1, \boldsymbol{x}_2 \dots \boldsymbol{x}_N\} = \{(u_1, v_1), (u_2, v_2) \dots (u_N, v_N)\}. \tag{11.5}$$

Für die Berechnung der meisten geometrischen Eigenschaften kann eine Region aus einer beliebigen Punktmenge bestehen und muss auch (im Unterschied zur Definition in Abschn. 11.1) nicht notwendigerweise zusammenhängend sein.

Umfang

Der Umfang (*perimeter*) einer Region $\mathcal{R}$ ist bestimmt durch die Länge ihrer äußeren Kontur, wobei $\mathcal{R}$ zusammenhängend sein muss. Wie Abb. 11.14 zeigt, ist bei der Berechnung die Art der Nachbarschaftsbeziehung zu beachten. Bei Verwendung der 4er-Nachbarschaft ist die Gesamtlänge der Konturesegmente (jeweils mit der Länge 1) i. Allg. größer als die tatsächliche Strecke.

Im Fall der 8er-Nachbarschaft wird durch Gewichtung der Horizontal- und Vertikalsegmente mit 1 und der Diagonalsegmente mit $\sqrt{2}$ eine gute Annäherung erreicht. Für eine Kontur mit dem 8-Chain Code $C_{\mathcal{R}} = (c_0, c_1, \dots c_{M-1})$ berechnet sich der Umfang daher in der Form

$$\mathit{Perimeter}(\mathcal{R}) = \sum_{i=0}^{M-1} \mathit{length}(c_i)\ , \tag{11.6}$$

wobei

$$\mathit{length}(c) = \begin{cases} 1 & \text{für } \ c = 0, 2, 4, 6, \\ \sqrt{2} & \text{für } \ c = 1, 3, 5, 7. \end{cases}$$

Bei dieser gängigen Form der Berechnung[8] wird allerdings die Länge des Umfangs gegenüber dem tatsächlichen Wert $U(\mathcal{R})$ systematisch überschätzt. Ein einfacher Korrekturfaktor von 0.95 erweist sich bereits bei relativ kleinen Regionen als brauchbar, d. h.

$$U(\mathcal{R}) \approx \mathit{Perimeter}_{\text{corr}}(\mathcal{R}) = 0.95 \cdot \mathit{Perimeter}(\mathcal{R}). \tag{11.7}$$

[8] Auch die im Analyze-Menü von ImageJ verfügbaren Messfunktionen verwenden diese Form der Umfangsberechnung.

Fläche

Die Fläche einer Region $\mathcal{R}$ berechnet sich einfach durch die Anzahl der enthaltenen Bildpunkte, d. h.

$$\mathit{Area}(\mathcal{R}) = N = |\mathcal{R}|. \tag{11.8}$$

Die Fläche einer zusammenhängenden Region (ohne Löcher) kann auch näherungsweise über ihre geschlossene Kontur, definiert durch M Koordinatenpunkte $(\boldsymbol{x}_0, \boldsymbol{x}_1, \ldots \boldsymbol{x}_{M-1})$, wobei $\boldsymbol{x}_i = (u_i, v_i)$, mit der Gauß'schen Flächenformel für Polygone) berechnet werden:

$$\mathit{Area}(\mathcal{R}) = \frac{1}{2} \cdot \left| \sum_{i=0}^{M-1} \Big(u_i \cdot v_{[(i+1) \bmod M]} - u_{[(i+1) \bmod M]} \cdot v_i \Big) \right| \tag{11.9}$$

Liegt die Kontur in Form eines Chain Codes $C_\mathcal{R} = (c_0, c_1, \ldots c_{M-1})$ vor, so kann die Fläche durch Expandieren von $C_\mathcal{R}$ in eine Folge von Konturpunkten, ausgehend von einem beliebigen Startpunkt (z. B. $(0, 0)$), ebenso mit Gl. 11.9 berechnet werden.

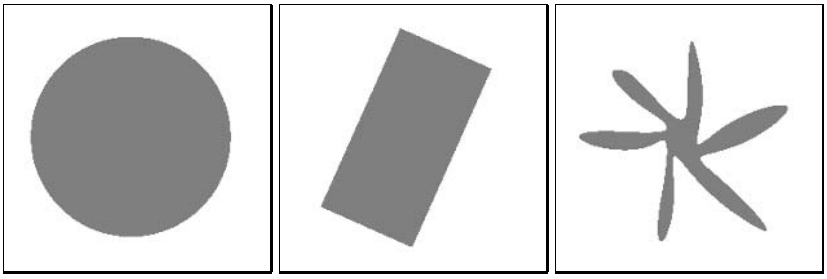
Einfache Eigenschaften wie Fläche und Umfang sind zwar (abgesehen von Quantisierungsfehlern) unbeeinflusst von Verschiebungen und Drehungen einer Region, sie verändern sich jedoch bei einer *Skalierung* der Region, wenn also beispielsweise ein Objekt aus verschiedenen Entfernungen aufgenommen wurde. Durch geschickte Kombination können jedoch neue Features konstruiert werden, die invariant gegenüber Translation, Rotation und Skalierung sind.

Kompaktheit und Rundheit

Unter „Kompaktheit" versteht man die Relation zwischen der Fläche einer Region und ihrem Umfang. Da der Umfang (*Perimeter*) U einer Region linear mit dem Vergrößerungsfaktor zunimmt, die Fläche (*Area*) A jedoch quadratisch, verwendet man das Quadrat des Umfangs in der Form A/U^2 zur Berechnung eines größenunabhängigen Merkmals. Dieses Maß ist invariant gegenüber Verschiebungen, Drehungen und Skalierungen und hat für eine kreisförmige Region mit beliebigem Durchmesser den Wert $\frac{1}{4\pi}$. Durch Normierung auf den Kreis ergibt sich daraus ein Maß für die „Rundheit" (*roundness*) oder „Kreisförmigkeit (*circularity*)

$$\mathit{Circularity}(\mathcal{R}) = 4\pi \cdot \frac{\mathit{Area}(\mathcal{R})}{\mathit{Perimeter}^2(\mathcal{R})}\,, \tag{11.10}$$

das für eine kreisförmige Region $\mathcal{R}$ den Maximalwert 1 ergibt und für alle übrigen Formen Werte im Bereich $[0, 1]$ (Abb. 11.15). Für eine absolute Schätzung der Kreisförmigkeit empfiehlt sich allerdings die Verwendung des korrigierten Umfangwerts aus Gl. 11.7, also



0.904 (1.001) | 0.607 (0.672) | 0.078 (0.086)

Abbildung 11.15
Circularity-Werte für verschiedene Regionsformen. Angegeben sind jeweils der Wert $Circularity(\mathcal{R})$ und in Klammern der korrigierte Wert $Circularity_{\text{corr}}(\mathcal{R})$ nach Gl. 11.10 bzw. 11.11.

$$Circularity_{\text{corr}}(\mathcal{R}) = 4\pi \cdot \frac{Area(\mathcal{R})}{Perimeter^2_{\text{corr}}(\mathcal{R})} \,. \tag{11.11}$$

In Abb. 11.15 sind die Werte für die Kreisförmigkeit nach Gl. 11.10 bzw. 11.11 für verschiedene Formen von Regionen dargestellt.

Bounding Box

Die Bounding Box einer Region $\mathcal{R}$ bezeichnet das minimale, achsenparallele Rechteck, das alle Punkte aus $\mathcal{R}$ einschließt:

$$BoundingBox(\mathcal{R}) = (u_{\min}, u_{\max}, v_{\min}, v_{\max}), \tag{11.12}$$

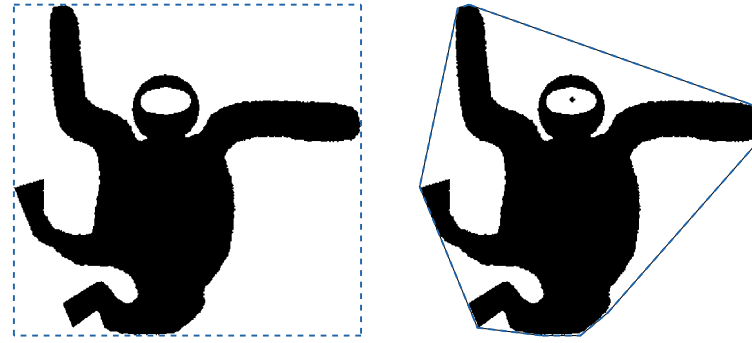
wobei $u_{\min}, u_{\max}$ und $v_{\min}, v_{\max}$ die minimalen und maximalen Koordinatenwerte aller Punkte $(u_i, v_i) \in \mathcal{R}$ in x- bzw. y-Richtung sind (Abb. 11.16 (a)).

Konvexe Hülle

Die konvexe Hülle (*convex hull*) ist das kleinste Polygon, das alle Punkte einer Region umfasst. Eine einfache Analogie ist die eines Nagelbretts, in dem für alle Punkte einer Region ein Nagel an der entsprechenden Position eingeschlagen ist. Spannt man nun ein elastisches Band rund um alle Nägel, dann bildet dieses die konvexe Hülle (Abb. 11.16 (b)). Sie kann z. B. mit dem *QuickHull*-Algorithmus [5] für N Konturpunkte mit einem Zeitaufwand von $\mathcal{O}(NH)$ berechnet werden, wobei H die Anzahl der resultierenden Polygonpunkte ist.[9]

Nützlich ist die konvexe Hülle beispielsweise zur Bestimmung der Konvexität oder der *Dichte* einer Region. Die *Konvexität* ist definiert als das Verhältnis zwischen der Länge der konvexen Hülle und dem Umfang der ursprünglichen Region. Unter *Dichte* versteht man hingegen das Verhältnis zwischen der Fläche der Region selbst und der Fläche der konvexen Hülle. Der *Durchmesser* wiederum ist die maximale Strecke zwischen zwei Knoten auf der konvexen Hülle.

[9] Zur Notation $\mathcal{O}()$ s. Anhang 1.3.



(a) (b)

Abbildung 11.16
Beispiel für *Bounding Box* (a) und konvexe Hülle (b) einer binären Bildregion.

11.4.3 Statistische Formeigenschaften

Bei der Berechnung von statistischen Formmerkmalen betrachten wir die Region $\mathcal{R}$ als Verteilung von Koordinatenpunkten im zweidimensionalen Raum. Statistische Merkmale können insbesondere auch für Punktverteilungen berechnet werden, die keine zusammengehörige Region bilden, und sind daher ohne vorherige Segmentierung einsetzbar. Ein wichtiges Konzept bilden in diesem Zusammenhang die so genannten *zentralen Momente* der Verteilung, die charakteristische Eigenschaften in Bezug auf deren Mittelpunkt bzw. *Schwerpunkt* ausdrücken.

Schwerpunkt

Den Schwerpunkt einer zusammenhängenden Region kann man sich auf einfache Weise so vorstellen, dass man die Region auf ein Stück Karton oder Blech zeichnet, ausschneidet und dann versucht, diese Form waagerecht auf einer Spitze zu balancieren. Der Punkt, an dem man die Region aufsetzen muss, damit dieser Balanceakt gelingt, ist der *Schwerpunkt* der Region.[10]

Der Schwerpunkt $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})$ einer binären (nicht notwendigerweise zusammenhängenden) Region berechnet sich als arithmetischer Mittelwert der Koordinaten in x- und y-Richtung, d. h.

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u \qquad \text{und} \qquad \bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} v \,. \tag{11.13}$$

Momente

Die Formulierung für den Schwerpunkt einer Region in Gl. 11.13 ist nur ein spezieller Fall eines allgemeineren Konzepts aus der Statistik, der so genannten „Momente“. Insbesondere beschreibt der Ausdruck

[10] Vorausgesetzt der Schwerpunkt liegt nicht innerhalb eines Lochs in der Region liegt, was durchaus möglich ist.

$$m_{pq} = \sum_{(u,v)\in\mathcal{R}} I(u,v) \cdot u^p v^q \tag{11.14}$$

das (gewöhnliche) Moment der Ordnung p, q für eine diskrete (Bild-)Funktion $I(u,v) \in \mathbb{R}$, also beispielsweise für ein Grauwertbild. Alle nachfolgenden Definitionen sind daher – unter entsprechender Einbeziehung der Bildfunktion $I(u,v)$ – grundsätzlich auch für Regionen in Grauwertbildern anwendbar. Für zusammenhängende, binäre Regionen können Momente auch direkt aus den Koordinaten der Konturpunkte berechnet werden [75, S. 148].

Für den speziellen Fall eines Binärbilds $I(u,v) \in \{0,1\}$ sind nur die Vordergrundpixel mit $I(u,v) = 1$ in der Region $\mathcal{R}$ enthalten, wodurch sich Gl. 11.14 reduziert auf

$$m_{pq} = \sum_{(u,v)\in\mathcal{R}} u^p v^q \,. \tag{11.15}$$

So kann etwa die **Fläche** einer binären Region als Moment nullter Ordnung in der Form

$$\mathit{Area}(\mathcal{R}) = |\mathcal{R}| = \sum_{(u,v)\in\mathcal{R}} 1 = \sum_{(u,v)\in\mathcal{R}} u^0 v^0 = m_{00}(\mathcal{R}) \tag{11.16}$$

ausgedrückt werden bzw. der **Schwerpunkt** $\bar{\boldsymbol{x}}$ (Gl. 11.13) als

$$\bar{x} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u^1 v^0 = \frac{m_{10}(\mathcal{R})}{m_{00}(\mathcal{R})} \tag{11.17}$$

$$\bar{y} = \frac{1}{|\mathcal{R}|} \cdot \sum_{(u,v)\in\mathcal{R}} u^0 v^1 = \frac{m_{01}(\mathcal{R})}{m_{00}(\mathcal{R})} \tag{11.18}$$

Diese Momente repräsentieren also konkrete physische Eigenschaften einer Region. Insbesondere ist die Fläche m_{00} in der Praxis eine wichtige Basis zur Charakterisierung von Regionen und der Schwerpunkt $(\bar{x}, \bar{y})$ erlaubt die zuverlässige und (auf Bruchteile eines Pixelabstands) genaue Bestimmung der Position einer Region.

Zentrale Momente

Um weitere Merkmale von Regionen unabhängig von ihrer Lage, also invariant gegenüber Verschiebungen, zu berechnen, wird der in jeder Lage eindeutig zu bestimmende Schwerpunkt als Referenz verwendet. Anders ausgedrückt, man verschiebt den Ursprung des Koordinatensystems an den Schwerpunkt $\bar{\boldsymbol{x}} = (\bar{x}, \bar{y})$ der Region und erhält dadurch die so genannten *zentralen* Momente der Ordnung p, q:

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} I(u,v) \cdot (u-\bar{x})^p \cdot (v-\bar{y})^q \tag{11.19}$$

Für Binärbilder (mit $I(u,v) = 1$ innerhalb der Region $\mathcal{R}$) reduziert sich Gl. 11.19 auf

$$\mu_{pq}(\mathcal{R}) = \sum_{(u,v)\in\mathcal{R}} (u - \bar{x})^p \cdot (v - \bar{y})^q \,. \tag{11.20}$$

Normalisierte zentrale Momente

Die Werte der zentralen Momente sind naturgemäß von der absoluten Größe der Region abhängig, da diese sich in den Distanzen aller Punkte vom Schwerpunkt direkt manifestiert. So multiplizieren sich bei einer gleichförmigen Vergrößerung einer Form um den Faktor $s \in \mathbb{R}$ die zentralen Momente mit dem Faktor

$$s^{(p+q+2)} \tag{11.21}$$

und man erhält größeninvariante (normalisierte) Momente durch Normierung mit dem Kehrwert der entsprechend potenzierten Fläche $\mu_{00} = m_{00}$ in der Form

$$\bar{\mu}_{pq}(\mathcal{R}) = \mu_{pq} \cdot \Big(\frac{1}{\mu_{00}(\mathcal{R})}\Big)^{(p+q+2)/2} \,, \tag{11.22}$$

für $(p+q) \geq 2$ [48, S. 529].

Prog. 11.3 zeigt eine direkte (*brute force*) Umsetzung der Berechnung von gewöhnlichen, zentralen und normalisierten Momenten in Java für binäre Bilder (`BACKGROUND = 0`). Dies ist nur zur Verdeutlichung gedacht und es sind natürlich weitaus effizientere Implementierungen möglich (z. B. in [50]).

11.4.4 Momentenbasierte geometrische Merkmale

Während die normalisierten Momente auch direkt zur Charakterisierung von Regionen verwendet werden können, sind einige interessante und geometrisch unmittelbar relevante Merkmale auf elegante Weise aus den Momenten ableitbar.

Orientierung

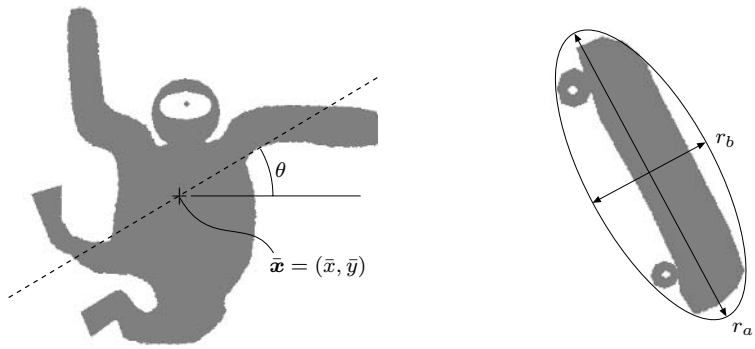
Die Orientierung bezeichnet die Richtung der Hauptachse, also der Achse, die durch den Schwerpunkt und entlang der größten Ausdehnung einer Region verläuft (Abb. 11.17 (a)). Dreht man die durch die Region beschriebene Fläche um ihre Hauptachse, so weist diese das geringste Trägheitsmoment aller möglichen Drehachsen auf. Wenn man etwa einen Bleistift zwischen beiden Händen hält und um seine Hauptachse (entlang der Bleistiftmine) dreht, dann treten wesentlich geringere Trägheitskräfte auf als beispielsweise bei einer propellerartigen Drehung quer zur Hauptachse (Abb. 11.18). Sofern die Region überhaupt eine

Programm 11.3
Beispiel für die direkte Berechnung von Momenten in Java. Die Methoden `moment()`, `centralMoment()` und `normalCentralMoment()` berechnen für ein Binärbild die Momente m_{pq}, μ_{pq} bzw. $\bar{\mu}_{pq}$ (Gl. 11.15, 11.20, 11.22).

```
1  import ij.process.ImageProcessor;
2
3  public class Moments {
4    static final int BACKGROUND = 0;
5
6    static double moment(ImageProcessor ip,int p,int q) {
7      double Mpq = 0.0;
8      for (int v = 0; v < ip.getHeight(); v++) {
9        for (int u = 0; u < ip.getWidth(); u++) {
10         if (ip.getPixel(u,v) != BACKGROUND) {
11           Mpq += Math.pow(u, p) * Math.pow(v, q);
12         }
13       }
14     }
15     return Mpq;
16   }

17   static double centralMoment(ImageProcessor ip,int p,int q)
18     {
19     double m00 = moment(ip, 0, 0); // region area
20     double xCtr = moment(ip, 1, 0) / m00;
21     double yCtr = moment(ip, 0, 1) / m00;
22     double cMpq = 0.0;
23     for (int v = 0; v < ip.getHeight(); v++) {
24       for (int u = 0; u < ip.getWidth(); u++) {
25         if (ip.getPixel(u,v) != BACKGROUND) {
26           cMpq +=
27             Math.pow(u - xCtr, p) *
28             Math.pow(v - yCtr, q);
29         }
30       }
31     }
32     return cMpq;
33   }

34   static double normalCentralMoment
35                         (ImageProcessor ip,int p,int q) {
36     double m00 = moment(ip, 0, 0);
37     double norm = Math.pow(m00, (double)(p + q + 2) / 2);
38     return centralMoment(ip, p, q) / norm;
39   }
40 }
```



(a) (b)

Abbildung 11.17
Orientierung und Exzentrizität. Die Hauptachse einer Region läuft durch ihren Schwerpunkt $\bar{\boldsymbol{x}}$ unter dem Winkel θ (a). Die Exzentrizität (b) ist ein Maß für das Seitenverhältnis (r_a/r_b) der kleinsten umhüllenden Ellipse, deren längere Achse parallel zur Hauptachse der Region liegt.

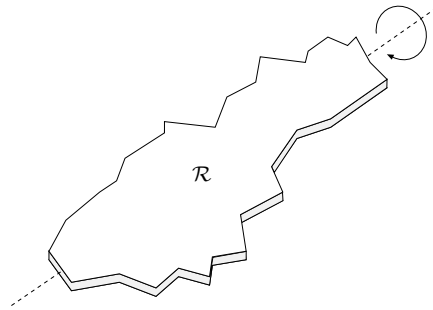


Abbildung 11.18
Hauptachse einer Region. Die Drehung einer länglichen Region $\mathcal{R}$ (interpretiert als physischer Körper) um ihre Hauptachse verursacht die geringsten Trägheitskräfte aller möglichen Achsen.

Orientierung aufweist ($\mu_{20}(\mathcal{R}) \neq \mu_{02}(\mathcal{R})$), ergibt sich die Richtung θ der Hauptachse aus den zentralen Momenten μ_{pq} als[11]

$$\theta(\mathcal{R}) = \frac{1}{2}\tan^{-1}\left(\frac{2\cdot\mu_{11}(\mathcal{R})}{\mu_{20}(\mathcal{R})-\mu_{02}(\mathcal{R})}\right) . \qquad (11.23)$$

Exzentrizität

Ähnlich wie die *Richtung* der Hauptachse lässt sich auch die *Länglichkeit* der Region über die Momente bestimmen. Das Merkmal der Exzentrizität (*eccentricity* oder *elongation*) kann man sich am einfachsten so vorstellen, dass man eine Region so lange dreht, bis sich eine Bounding Box oder umhüllende Ellipse mit maximalem Seitenverhältnis ergibt (Abb. 11.17 (b)).[12] Aus dem Verhältnis zwischen der Breite und der Länge der resultierenden Bounding Box bzw. Ellipse können unterschiedliche Maße für die Exzentrizität der Region abgeleitet werden [4, 49] (s. auch Aufg.

[11] Siehe Anhang B.1.6 bezüglich der Winkelberechnung in Java mit `Math.atan2()`.

[12] Dieser Vorgang wäre in der Praxis natürlich allein wegen der vielfachen Bildrotation relativ rechenaufwendig.

11.11). In [48, S. 531] etwa wird auf Basis der zentralen Momente μ_{pq} ein Exzentrizitätsmaß in der Form

$$Eccentricity(\mathcal{R}) = \frac{\big[\mu_{20}(\mathcal{R}) - \mu_{02}(\mathcal{R})\big]^2 + 4 \cdot \big[\mu_{11}(\mathcal{R})\big]^2}{\big[\mu_{20}(\mathcal{R}) + \mu_{02}(\mathcal{R})\big]^2} \tag{11.24}$$

definiert. Dieses Maß ergibt Werte im Bereich $[0, 1]$. Ein rundes Objekt weist demnach eine Exzentrizität von 0 auf, ein extrem langgezogenes Objekt den Wert 1, also entgegengesetzt zur „Rundheit“ in Gl. 11.10. Die Berechnung der Exzentrizität ist allerdings in diesem Fall ohne explizite Kenntnis des Umfangs und der Fläche möglich und somit auch für nicht zusammenhängende Regionen anwendbar.

Invariante Momente

Normalisierte zentrale Momente sind zwar unbeeinflusst durch eine Translation oder gleichförmige Skalierung einer Region, verändern sich jedoch im Allgemeinen bei einer *Rotation* des Bilds. Ein klassisches Beispiel zur Lösung dieses Problems durch geschickte Kombination einfacher Merkmale sind die nachfolgenden, als „Hu's Momente“ [39] bekannten Größen:[13]

$$\begin{aligned}
H_1 &= \bar{\mu}_{20} + \bar{\mu}_{02} \qquad (11.25)\\
H_2 &= (\bar{\mu}_{20} - \bar{\mu}_{02})^2 + 4\,\bar{\mu}_{11}^2\\
H_3 &= (\bar{\mu}_{30} - 3\,\bar{\mu}_{12})^2 + (3\,\bar{\mu}_{21} - \bar{\mu}_{03})^2\\
H_4 &= (\bar{\mu}_{30} + \bar{\mu}_{12})^2 + (\bar{\mu}_{21} + \bar{\mu}_{03})^2\\
H_5 &= (\bar{\mu}_{30} - 3\,\bar{\mu}_{12}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot \big[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3(\bar{\mu}_{21} + \bar{\mu}_{03})^2\big] +\\
&\quad (3\,\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot \big[3\,(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\big]\\
H_6 &= (\bar{\mu}_{20} - \bar{\mu}_{02}) \cdot \big[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\big] +\\
&\quad 4\,\bar{\mu}_{11} \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03})\\
H_7 &= (3\,\bar{\mu}_{21} - \bar{\mu}_{03}) \cdot (\bar{\mu}_{30} + \bar{\mu}_{12}) \cdot \big[(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - 3\,(\bar{\mu}_{21} + \bar{\mu}_{03})^2\big] +\\
&\quad (3\,\bar{\mu}_{12} - \bar{\mu}_{30}) \cdot (\bar{\mu}_{21} + \bar{\mu}_{03}) \cdot \big[3\,(\bar{\mu}_{30} + \bar{\mu}_{12})^2 - (\bar{\mu}_{21} + \bar{\mu}_{03})^2\big]
\end{aligned}$$

In der Praxis wird allerdings meist der Logarithmus der Ergebnisse (also $\log(H_k)$) verwendet, um den ansonsten sehr großen Wertebereich zu reduzieren. Diese Features werden auch als „*moment invariants*“ bezeichnet, denn sie sind weitgehend invariant unter Translation, Skalierung sowie Rotation. Sie sind auch auf Ausschnitte von Grauwertbildern anwendbar, Beispiele dafür finden sich etwa in [30, S. 517].

[13] Das Argument für die Region ($\mathcal{R}$) wird in Gl. 11.25 zur besseren Lesbarkeit weggelassen, die erste Zeile würde also vollständig lauten: $H_1(\mathcal{R}) = \bar{\mu}_{20}(\mathcal{R}) + \bar{\mu}_{02}(\mathcal{R})$, usw.

Programm 11.4
Berechnung von horizontaler und vertikaler Projektion. Die `run()`-Methode für ein ImageJ-Plugin (`ip` ist vom Typ `ByteProcessor` oder `ShortProcessor`) berechnet in einem Bilddurchlauf beide Projektionen als eindimensionale Arrays (`horProj`, `verProj`) mit Elementen vom Typ `long`.

```
1  public void run(ImageProcessor ip) {
2    int M = ip.getWidth();
3    int N = ip.getHeight();
4    long[] horProj = new long[N];
5    long[] verProj = new long[M];
6    for (int v = 0; v < N; v++) {
7      for (int u = 0; u < M; u++) {
8        int p = ip.getPixel(u, v);
9        horProj[v] += p;
10       verProj[u] += p;
11     }
12   }
13   // use projections horProj, verProj now
14   // ...
15 }
```

11.4.5 Projektionen

Projektionen von Bildern sind eindimensionale Abbildungen der Bilddaten, üblicherweise parallel zu den Koordinatenachsen. In diesem Fall ist die horizontale bzw. vertikale Projektion für ein Bild $I(u, v)$, mit $0 \le u < M$, $0 \le v < N$, definiert als

$$P_{\text{hor}}(v_0) = \sum_{u=0}^{M-1} I(u, v_0) \qquad \text{für } 0 < v_0 < N, \tag{11.26}$$

$$P_{\text{ver}}(u_0) = \sum_{v=0}^{N-1} I(u_0, v) \qquad \text{für } 0 < u_0 < M. \tag{11.27}$$

Die *horizontale* Projektion $P_{\text{hor}}(v_0)$ (Gl. 11.26) bildet also die Summe der Pixelwerte der Bild*zeile* v_0 und hat die Länge N, die der Höhe des Bilds entspricht. Umgekehrt ist die *vertikale* Projektion P_{ver} (der Länge M) die Summe aller Bildwerte in der *Spalte* u_0 (Gl. 11.27). Im Fall eines Binärbilds mit $I(u, v) \in 0, 1$ enthält die Projektion die Anzahl der Vordergrundpixel in der zugehörigen Bildzeile bzw. -spalte.

Prog. 11.4 zeigt eine einfache Implementierung der Projektionsberechnung als `run()`-Methode eines ImageJ-Plugin, wobei beide Projektionen in einem Bilddurchlauf berechnet werden. Für die Projektionen werden Arrays vom Typ `long` (64-Bit-Integer) verwendet, um auch bei großen Bildern einen arithmetischen Überlauf zu vermeiden.

Projektionen in Richtung der Koordinatenachsen sind beispielsweise zur schnellen Analyse von strukturierten Bildern nützlich, wie etwa zur Isolierung der einzelnen Zeilen in Textdokumenten oder auch zur Trennung der Zeichen innerhalb einer Textzeile (Abb. 11.19). Grundsätzlich sind aber Projektionen auf Geraden mit beliebiger Orientierung möglich, beispielsweise in Richtung der Hauptachse einer gerichteten Bildregion (Gl. 11.23). Nimmt man den Schwerpunkt der Region (Gl. 11.13) als Re-

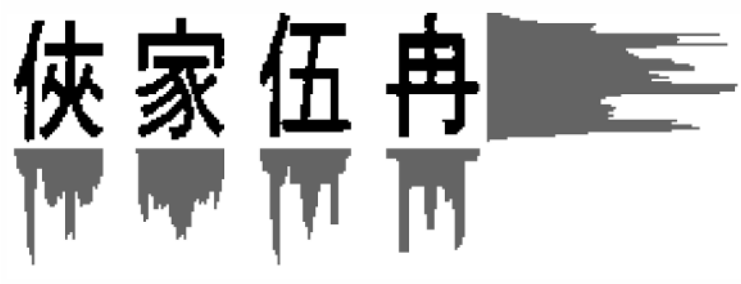


Abbildung 11.19
Beispiel für die horizontale und vertikale Projektion eines Binärbilds.

ferenz entlang der Hauptachse, so erhält man eine weitere, rotationsinvariante Beschreibung der Region in Form des Projektionsvektors.

11.4.6 Topologische Merkmale

Topologische Merkmale beschreiben nicht explizit die Form einer Region, sondern strukturelle Eigenschaften, die auch unter stärkeren Bildverformungen unverändert bleiben. Dazu gehört auch die Eigenschaft der Konvexität einer Region, die sich durch Berechnung ihrer konvexen Hülle (Abschn. 11.4.2) bestimmen lässt.

Ein einfaches und robustes topologisches Merkmal ist die *Anzahl der Löcher* $N_L(\mathcal{R})$, die sich aus der Berechnung der inneren Konturen einer Region ergibt, wie in Abschn. 11.2.2 beschrieben. Umgekehrt kann eine nicht zusammenhängende Region, wie beispielsweise der Buchstabe „**i**“, aus mehreren Komponenten bestehen, deren Anzahl ebenfalls als Merkmal verwendet werden kann.

Ein davon abgeleitetes Merkmal ist die so genannte *Euler-Zahl* N_E, das ist die Anzahl der zusammenhängenden Regionen N_R abzüglich der Anzahl ihrer Löcher N_L, d. h.

$$N_E(\mathcal{R}) = N_R(\mathcal{R}) - N_L(\mathcal{R}). \tag{11.28}$$

Bei nur *einer* zusammenhängenden Region ist dies einfach $1 - N_L$. So gilt für die Ziffer „**8**“ beispielsweise $N_L = 1 - 2 = -1$ oder $N_L = 1 - 1 = 0$ für den Buchstaben „**D**“.

Topologische Merkmale werden oft in Kombination mit numerischen Features zur Klassifikation verwendet, etwa für die Zeichenerkennung (*optical character recognition*, OCR) [13].

11.5 Aufgaben

Aufg. 11.1. Simulieren Sie manuell den Ablauf des *Flood-fill*-Verfahrens in Prog. 11.1 (*depth-first* und *breadth-first*) anhand einer Region des folgenden Bilds, beginnend bei der Startkoordinate $(5, 1)$:

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	1	1	0	1	0
0	1	1	1	1	1	1	0	0	1	0	0	1	0
0	0	0	0	1	0	1	0	0	0	0	0	1	0
0	1	1	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	1	1	1	1	1	1	1	1	1	0
0	1	1	0	0	0	1	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

0 Hintergrund

1 Vordergrund

Aufg. 11.2. Bei der Implementierung des *Flood-fill*-Verfahrens (Prog. 11.1) werden bei jedem bearbeiteten Pixel alle seine Nachbarn im *Stack* bzw. in der *Queue* vorgemerkt, unabhängig davon, ob sie noch innerhalb des Bilds liegen und Vordergrundpixel sind. Man kann die Anzahl der im *Stack* bzw. in der *Queue* zu speichernden Knoten reduzieren, indem man jene Nachbarpixel ignoriert, die diese Bedingungen nicht erfüllen. Modifizieren Sie die *Depth-first*- und *Breadth-first*-Variante in Prog. 11.1 entsprechend und vergleichen Sie die resultierenden Laufzeiten.

Aufg. 11.3. Implementieren Sie ein ImageJ-Plugin, das auf ein Grauwertbild eine Lauflängenkodierung (Abschn. 11.3.2) anwendet, das Ergebnis in einer Datei ablegt und ein zweites Plugin, das aus dieser Datei das Bild wieder rekonstruiert.

Aufg. 11.4. Berechnen Sie den erforderlichen Speicherbedarf zur Darstellung einer Kontur mit 1000 Punkten auf folgende Arten: (a) als Folge von Koordinatenpunkten, die als Paare von `int`-Werten dargestellt sind; (b) als 8-Chain Code mit Java-`byte`-Elementen; (c) als 8-Chain Code mit nur 3 Bits pro Element.

Aufg. 11.5. Implementieren Sie eine Java-Klasse zur Beschreibung von binären Bildregionen mit Chain Codes. Entscheiden Sie selbst, ob Sie einen absoluten oder differentiellen Chain Code verwenden. Die Implementierung soll in der Lage sein, geschlossene Konturen als Chain Codes zu kodieren und auch wieder zu rekonstruieren.

Aufg. 11.6. Durch Berechnung der konvexen Hülle kann auch der maximale Durchmesser (max. Abstand zwischen zwei beliebigen Punkten) einer Region auf einfache Weise berechnet werden. Überlegen Sie sich ein alternatives Verfahren, das diese Aufgabe ohne Verwendung der komplexen Hülle löst. Ermitteln Sie den Zeitaufwand Ihres Algorithmus in Abhängigkeit zur Anzahl der Punkte in der Region.

Aufg. 11.7. Implementieren Sie den Vergleich von Konturen auf Basis von „Shape Numbers“ (Gl. 11.3). Entwickeln Sie dafür eine Metrik, welche die Distanz zwischen zwei normalisierten Chain Codes misst. Stellen Sie fest, ob und unter welchen Bedingungen das Verfahren zuverlässig arbeitet.

Aufg. 11.8. Entwerfen Sie einen Algorithmus, der aus einer als 8-Chain Code gegebenen Kontur auf der Basis von Gl. 11.9 die Fläche der zugehörigen Region berechnet. Welche Abweichungen sind gegenüber der tatsächlichen Fläche der Region (Anzahl der Pixel) zu erwarten?

Aufg. 11.9. Skizzieren Sie Beispiele von binären Regionen, bei denen der Schwerpunkt selbst nicht in der Bildregion liegt.

Aufg. 11.10. Implementieren Sie die Momenten-Features von Hu (Gl. 11.25) und überprüfen Sie deren Eigenschaften unter Skalierung und Rotation anhand von Binär- und Grauwertbildern.

Aufg. 11.11. Für die Exzentrizität einer Region (Gl. 11.24) gibt es alternative Formulierungen, zum Beispiel [49, S. 394]

$$Eccentricity_2(\mathcal{R}) = \frac{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}}{m_{00}} \quad \text{oder} \tag{11.29}$$

$$Eccentricity_3(\mathcal{R}) = \frac{\mu_{20} + \mu_{02} + \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}}{\mu_{20} + \mu_{02} - \sqrt{(\mu_{20} - \mu_{02})^2 + 4 \cdot \mu_{11}^2}} \,. \tag{11.30}$$

Realisieren Sie alle drei Varianten (einschließlich Gl. 11.24) und vergleichen Sie die Ergebnisse anhand geeigneter Regionsformen.

Aufg. 11.12. Die Java-Methode in Prog. 11.4 berechnet die Projektionen eines Bilds in horizontaler und vertikaler Richtung. Bei der Verarbeitung von Dokumentenvorlagen werden u. a. auch Projektionen in Diagonalrichtung eingesetzt. Implementieren Sie diese Projektionen und überlegen Sie, welche Rolle diese in der Dokumentenverarbeitung spielen könnten.